QoS-Aware Management of Monotonic Service Orchestrations

Albert Benveniste, Claude Jard, Ajay Kattepur, Sidney Rosario, John A. Thywissen

▶ **To cite this version:**

Albert Benveniste, Claude Jard, Ajay Kattepur, Sidney Rosario, John A. Thywissen. QoS-Aware Management of Monotonic Service Orchestrations. Formal Methods in System Design, 2014, 44 (1), pp.1-43. 10.1007/s10703-013-0191-7 . hal-00840362

HAL Id: hal-00840362
https://hal.science/hal-00840362

Submitted on 2 Jul 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Formal Methods in System Design

QoS-Aware Management of Monotonic Service Orchestrations

--Manuscript Draft--

Manuscript Number:	FORM-D-12-00304R1
Full Title:	QoS-Aware Management of Monotonic Service Orchestrations
Article Type:	Manuscript
Keywords:	Web Services; Service orchestrations; Quality of Service Algebra; Probabilistic Models.
Corresponding Author:	Albert Benveniste Inria Rennes, FRANCE
Corresponding Author Secondary Information:	
Corresponding Author's Institution:	Inria
Corresponding Author's Secondary Institution:	
First Author:	Albert Benveniste
First Author Secondary Information:	
Order of Authors:	Albert Benveniste
	Claude Jard
	Ajay Kattepur
	Sidney Rosario
	John A. Thywissen
Order of Authors Secondary Information:	
Abstract:	We study QoS-aware management of service orchestrations, specifically for orchestrations having a data-dependent workflow. Our study supports multi-dimensional QoS. To capture uncertainty in performance and QoS, we provide support for probabilistic QoS. Under the above assumptions, orchestrations may be non-monotonic with respect to QoS, meaning that strictly improving the QoS of a service may strictly decrease the end-to-end QoS of the orchestration, an embarrassing feature for QoS-aware management. We study monotonicity and provide sufficient conditions for it. We then propose a comprehensive theory and methodology for monotonic orchestrations. Generic QoS composition rules are developed via a QoS Calculus, also capturing best service binding -- service discovery, however, is not within the scope of this work. Monotonicity provides the rationale for a contract-based approach to QoS-aware management. Although function and QoS cannot be separated in the design of complex orchestrations, we show that our framework supports separation of concerns by allowing the development of function and QoS separately and then weaving them together to derive the QoS-enhanced orchestration. Our approach is implemented on top of the Orc script language for specifying service orchestrations.

Powered by Editorial Manager® and Preprint Manager® from Aries Systems Corporation

Revised version of Manuscript D-12-00304
Answers to the reviewers

February 18, 2013

1 General answers

We thank the reviewers for their careful reading and criticism. Detailed answers to each reviewer are provided below. Main changes in this version are the following:

1. We must agree with the main criticism of reviewer #2. Indeed, having submitted this paper, we decided to re-investigate with Samy Abbes branching cells for nets with read arcs and found errors in the definitions of Sydney Rosario's thesis. This had impact on the definition of Procedure 1 and correcting it would result in a significant increase in complexity. Our formulation of Procedure 1 in the first submission was indeed incorrect as pointed out by reviewer #2, and we are very grateful for this. We have decided to follow the advise of this reviewer by restricting ourselves to the much simpler case of free choice nets. This adequately puts the focus on the handling of the QoS, which remains the main contribution of this paper.

2. Since our submission, the development of the QoS-Orc tool has been pursued and we have updated the second Appendix to account for the resulting changes and we provide more details.

3. We have shifted the related work later in the paper.

2 Detailed answers

2.1 Reviewer #1

We thank you for your careful reading.

- *Intro: The authors should provide more discussion about the relations of this paper with [15,39,40].* Our paper states on page 7: "This agenda was developed by a subgroup of authors of this paper in [15, 39, 40], for the restricted case of latency. In this paper we extend our previous work on latency-aware management of composite services to generic, possibly multi-dimensional, QoS." The extension is non-trivial because it covers generic, partially ordered, QoS. We do not see what we should say more.

- *[Pag. 8, Line 32/33] "non-probabilistic" excedes the page margin.* Does not apply to the revised version.

- *[Pag. 13, Line 34] "the left most token"* $\rightarrow$ *the token in the left most place (In Fig. 4).* Modified as requested.

- nition). Modified as requested.

- *[Pag. 18, Line. 10] "1/"* $\rightarrow$ *"(1)" , "2/"* $\rightarrow$ *"(2)".* Modified as requested.

- *[Pag. 20, Def. 2] "daemons" have the same role of "schedulers" in the terminology of Lynch, Segala. Please clarify.* A footnote has been added when introducing daemons. Our use is wider than the one you mention but the relation indeed holds.

2.2 Reviewer #2

We thank you very much for your highly valuable reading.

- *Page 4, line 26-27 "In general we believe that the term Quality of Service presupposes that better QoS is indeed better overall". Is that just what you believe? Or is it what practitioners in the field implicitly apply?* Changed the wording to "It is commonly understood".

- *Page 16, line 38-39. I think that* D^0 *is not the empty set, but the singleton of the empty tuple.* Replaced tuples by sets, so the problem disappears.

- *Page 38, line 29: what does it mean that k-dagger cannot occur?* Modified; we hope it is clarified.

*Manuscript
Click here to download Manuscript: FMSD_springer_revised.tex
Click here to view linked References

Formal Methods in Systems Design manuscript No.
(will be inserted by the editor)

QoS-Aware Management of Monotonic Service Orchestrations

Albert Benveniste · Claude Jard · Ajay Kattepur · Sidney Rosario · John A. Thywissen

Received: date / Accepted: date

Abstract We study QoS-aware management of service orchestrations, specifically for orchestrations having a data-dependent workflow. Our study supports *multi-dimensional* QoS. To capture uncertainty in performance and QoS, we provide support for *probabilistic* QoS. Under the above assumptions, orchestrations may be *non-monotonic* with respect to QoS, meaning that strictly improving the QoS of a service may strictly decrease the end-to-end QoS of the orchestration, an embarrassing feature for QoS-aware management. We study monotonicity and provide sufficient conditions for it. We then propose a comprehensive theory and methodology for monotonic orchestrations. Generic QoS composition rules are developed via a *QoS Calculus*, also capturing best service binding—service discovery, however, is not within the scope of this work. Monotonicity provides the rationale for a *contract-based* approach to QoS-aware management. Although function and QoS cannot be separated in the design of complex orchestrations, we show that our framework supports separation of concerns by allowing the development of function and QoS separately and then "weaving" them together to derive the QoS-enhanced orchestration. Our approach is implemented on top of the Orc script language for specifying service orchestrations.

A. Benveniste and A. Kattepur
DistribCom team at INRIA Rennes,
Campus Universitaire de Beaulieu, 35042 Rennes CEDEX, France.
E-mail: Firstname.Lastname@inria.fr

C. Jard
Department of Computer Science, Université de Nantes, LINA-Atlanstic
rue de la Houssinière, 44322 Nantes CEDEX 3, France
E-mail: Claude.Jard@univ-nantes.fr

J.A. Thywissen
Department of Computer Science, The University of Texas at Austin,
1 University Station, Austin, Texas 78712, USA.
This work started when S. Rosario was with U.T. Austin.
E-mail: sidney.rosario@gmail.com, jthywiss@cs.utexas.edu

Keywords Web Services · QoS · Algebra · Probabilistic Models

1 Introduction

Service Oriented Computing is a paradigm suited to wide area computing, where services can be dynamically selected and combined to offer a new service [17]. To enable service selection and binding, services expose both functional and Quality of Service (QoS) properties. Service selection can thus occur on the basis of both types of properties. In particular, service selection among a pool of functionally substitutable services can be performed based on QoS. Therefore, models of composite services should involve policies for QoS-based service selection [17].

Quite often, several dimensions for QoS must be handled (e.g., timing performance, availability, cost), leading to the consideration of multi-dimensional QoS. Consequently, QoS domains should be partially, not totally ordered. For simple policies, QoS guarantees exposed by the service or expected by the user are typically stated as fixed bounds. QoS is, however, generally subject to uncertainties, due to the numerous hidden sources of nondeterminism (servers, OS, queues, and network infrastructure). Therefore, a number of authors have agreed that QoS should be characterized in probabilistic terms [27, 39, 40, 53].

To illustrate the issues behind the QoS aware management of composite services, consider the following toy example, for which we first present a simple form and then develop some variations. Fig. 1(a) depicts a simple orchestration for a travel agent. The user enters the location of a place to visit. Two Airline services are invoked in parallel with the one offering "best" cost being selected. Next, two Hotel reservation services are invoked and selection occurs on the basis of cost and category, seen both as QoS dimensions. The selection on cost/category may be done through lexicographic or weighted ordering. The results are presented as an invoice. This orchestration exhibits a control flow that is independent from the circulated data. It has a two-dimensional QoS, with the two dimensions being cost and hotel category. Observe that the two QoS dimensions in this example are correlated.

This diagram is reformulated into that of Fig. 1(b), to be interpreted as a Petri net, where rectangles figure transitions and rounded rectangles figure places. Each query to the orchestration is modeled by a token traversing the input transition. Upon entering the first place, the transition to traverse must be chosen. This choice is based on best cost among offers by airline companies. Subsequently, the token enters the second place, where choice among different hotel booking services (shown as transitions) occurs based on both cost and category. This alternative Petri net description is a formalization of the previous description. We shall follow this Petri net modeling style hereinafter.

Fig. 2 shows a variation of Fig. 1(b) with a control flow dependent on both returned *data* and *QoS* values. A loop is introduced in the decision process that checks if the total Cost is within the budget and can ask the user to specify preferences again. The presentment of the Invoice is guarded by a timer. The

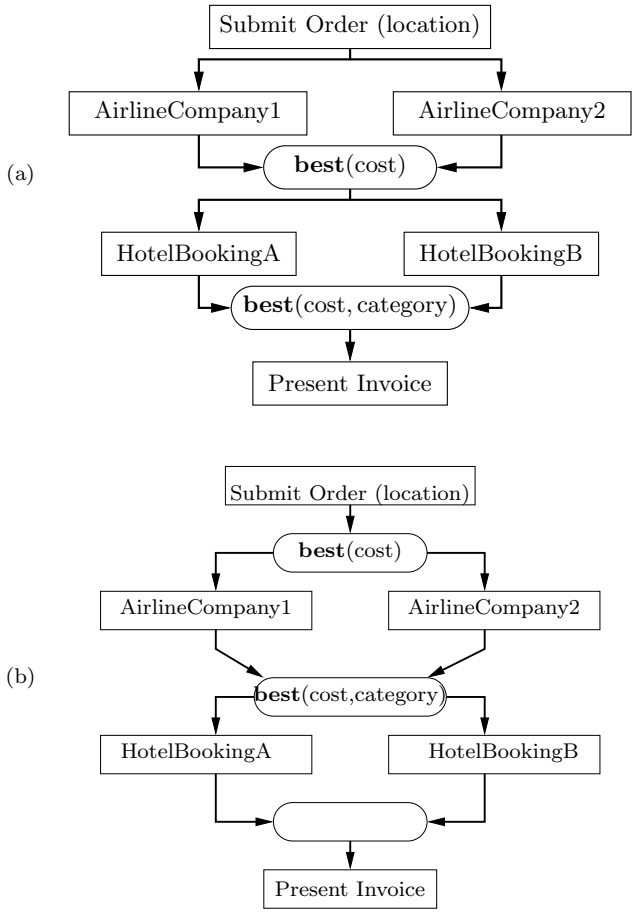

Fig. 1 TravelAgent1: Simple travel agent; (a) informal diagram, and (b) Petri net form, where rectangles figure transitions and rounded rectangles figure places. This orchestration has a data-independent workflow.

choice at the place labeled with "best(latency)" depends on which subsequent transition fires first. Thus, if the Invoice is ready before Timeout occurs, then it is emitted, otherwise a "timeout" message is returned. This timeout mechanism ensures that the loop terminates within a pre-specified time bound, possibly with a failure. This orchestration has a three-dimensional QoS, with the dimensions being cost, hotel category, and latency (due to timers). We now review some important issues in Service Oriented Computing.

Monotonicity and Consequences for Management: A basic assumption underpinning the management of composite services is that QoS improvements in component services can only be better for the composite service. For example, once service selection in QoS-based design has been performed, a selected service is not expected to get deselected if it improves its QoS performance.

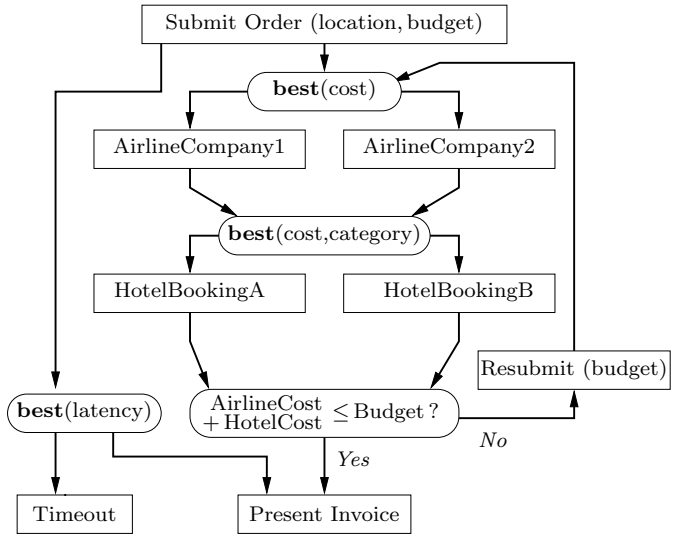

Fig. 2 TravelAgent2: A variation of TravelAgent1 having a data-dependent workflow.

Similarly, once services have been selected on the basis of QoS performance, reconfiguration will not occur unless some requested service's performance degrades or some non-selected service's performance improves. Finally, QoS monitoring consists of checking components for possible degradations in QoS. It is commonly understood that the term "Quality of Service" presupposes that "better QoS is indeed better overall". In other words, the better the involved services[1] perform, the better the composite service performs. This general property is important, so we give it a name—*monotonicity*. If a composite service fails to be monotonic, the common understanding of QoS is no longer valid and negotiations between the service provider and service requester regarding QoS issues become nearly unmanageable. We see monotonicity as a highly desirable feature, so we make it a central topic of this work.

Monotonicity always holds for orchestrations having a data-independent workflow—the orchestration shown in Fig. 1 is an example. A careful inspection shows that the orchestration of Fig. 2, which possesses a data-dependent workflow, is also monotonic.

However, monotonicity is not always satisfied. Consider the example in Fig. 3. This orchestration performs late binding of service by deciding online and based on the cost of the airline ticket, which company to select. The two companies then propose different sets of hotels, shown by the two steps HotelBookingA/B. Let c_1 and c_2 be the cost of ticket for Companies 1 and 2, and $h(c)$ be the optimum cost of the hotel booking if company c was selected. Suppose $c_1 < c_2$ and $h(c_1) < h(c_2)$ both hold. Then, the left branch is

[1] *Involved services* include all services that can potentially be requested by the composite service. For example, if the composite service involves an if-then-else branch, only one branch will actually be executed, but both are involved in the composite service.

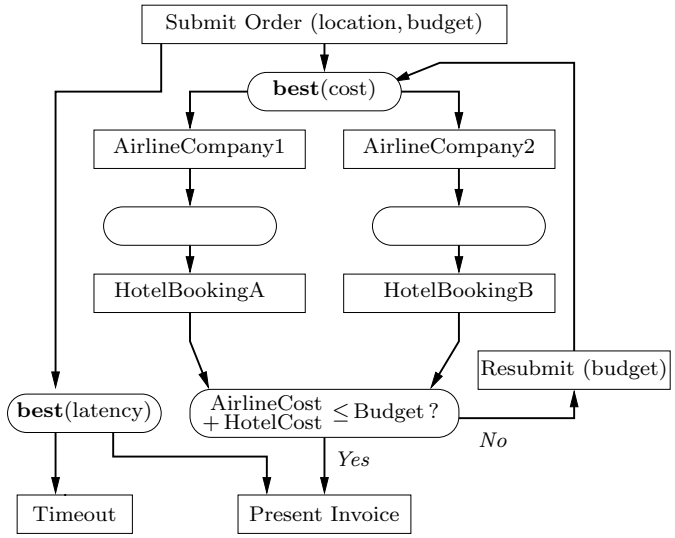

Fig. 3 TravelAgent3: A variation of TravelAgent2 lacking monotonicity.

preferred and yields a total cost QoS for the orchestration equal to $c_1 + h(c_1)$. Now, suppose Company 2 improves its offer beyond Company 1: $c_2 < c_1$. Then, the right branch will be selected and total cost $c_2 + h(c_2)$ will result. Now, it may very well be that $c_2 + h(c_2) > c_1 + h(c_1)$ still holds, meaning that the improvement in QoS of Company 2 has resulted in a degradation of total QoS. The orchestration of Fig. 3 is thus non-monotonic, despite it being a quite minor modification of TravelAgent2. Differences in "local" versus "global" optimization due to lack of monotonicity were identified in [8,11,51]—"monotonicity" was not mentioned in the referred works but the concept was identified.

Examples such as TravelAgent3 can be easily specified using the standard language BPEL for orchestrations and business processes. To summarize, this issue of monotonicity is essential. However, it seems underestimated in the literature on Web services, with the exception of [8, 11, 51], as our discussion of related work will show.

Assume that monotonicity is addressed, either by enforcing it, or by dealing with the lack of it. Then, new avenues for composite service management can be considered, by taking advantage of monotonicity:

(a) A called service that strictly improves its QoS cannot strictly worsen the QoS of the orchestration. Therefore, it is enough for the orchestration to monitor QoS degradations for each called service. Negotiations and penalties occur on the basis of understandable rules. If suitable, relations between the orchestration and its called services can rely on *QoS contracts*. It is then the duty of the orchestration (or of some third party) to monitor such contracts for possible violation.

(b) Since we build on a contract-based philosophy, the orchestration itself must be able to offer QoS contracts to its customers. This necessitates relating the contracts the orchestration has with its involved services to the overall contract it can offer to its customers. We refer to this as *contract composition*.
(c) Thanks to monotonicity, it is possible to perform QoS-based *late binding* of services by selecting, at run time, the best offer among a pool of compliant candidates—by "compliant" we mean candidates satisfying some given functional property.

Handling Probabilistic QoS: To handle uncertainty in QoS, probabilistic frameworks have been favored by a number of authors [18, 26, 27, 34, 39, 40, 44, 50, 51, 52, 53]. When the workflow of the orchestration is statically defined regardless of data, rules for composing QoS probability distributions of the called services have been proposed for various QoS domains [6, 8, 9, 10, 11, 19, 49]. Optimal service selection among different options has been solved by efficient optimization methods, by using, for example, Markov models [6, 18].

For orchestrations exhibiting data-dependent workflow or QoS values, however, such methods do not apply. The QoS-aware model of the orchestration combines *probability* and *non-determinism* — non-determinism arises from the data-dependent selection among alternatives. Markov models do not apply, and Markov Decision Process models must be considered instead. The successive data-dependent choices performed are referred to as the *scheduler* of the MDP. Optimization can then be stated in two different ways. In most approaches [18, 26, 27, 34, 44, 53], the scheduler itself is also randomized, thus resulting in a larger Markov model (assuming that sources of randomness are all independent). Alternatively, a max-min optimization can be performed, where the min is computed among the different service alternatives for a given fixed scheduler, and then the max over schedulers is computed. These methods have been widely used for off-line orchestration design. Optimal on-line service selection or binding is much more demanding. Mathematically speaking, this activity amounts to solving a *stochastic control* problem [7], in which, at each decision step, the expected remaining overall QoS is optimized and best decision is taken. Stochastic control is computationally demanding unless the considered orchestration is very small—this approach has not been considered in the literature.

In the previous paragraph, we have advocated the importance of monotonicity and have discussed its (good) consequences for QoS-aware management of composite services. Can we lift these considerations to probabilistic QoS? To compare random variables, *stochastic ordering* has been proposed in various forms and extensively used in the area of economics and operations research [36, 37, 46]. Using this concept, monotonicity was lifted to the probabilistic setting for the particular case of latency in [15]. Assuming that monotonicity can be lifted to the probabilistic setting for general QoS, the approach outlined in (a), (b), and (c) above becomes applicable and simple techniques can be developed for QoS aware service management based on con-

tracts. This agenda was developed by a subgroup of authors of this paper in [15, 39, 40], for the restricted case of latency.

Our Contribution: In this paper we extend our previous work on latency-aware management of composite services to generic, possibly multi-dimensional, QoS. In particular, QoS domains are no longer totally but only partially ordered, which causes significant increase in difficulty. Also, we take advantage of our formal approach to QoS management in developing a technique of *weaving* QoS aspects in the functional specification of a composite service. Our approach proceeds through the three steps (a), (b), and (c). Overall, we see our main contribution as being *a comprehensive and mathematically sound framework for contract-based QoS-aware management of composite services*, relying on monotonicity. This framework consists of the following.

An Abstract Algebraic Framework for QoS composition: As QoS composition is the primary building block of QoS-aware management, it is of interest to develop abstract algebraic composition rules. We propose such an *abstract algebraic framework* encompassing key properties of QoS domains and capturing how the QoS of the orchestration follows from combining QoS contributions by each requested service. This algebraic framework relies on an abstract dioid[2] $(\mathbb{D}, \max, \oplus)$, where $\mathbb{D}$ is the (possibly multi-dimensional) QoS domain. The abstract addition of the dioid identifies with the "max" operation associated with the partial order of the QoS domain; it captures both the preference among services in competition and the cost of synchronizing the return of several services requested in parallel. The increment in QoS caused by the different service calls is captured by the abstract multiplication of the dioid, here denoted by $\oplus$. A dioid framework for QoS was already proposed in [12,13,14,16,17,48]. With comparison to the above references, we propose in addition a new *competition operator* that must be considered when performing late binding; this competition operator captures the additional cost of the on-line comparison of the QoS within a pool of competing services. We show how our abstract algebraic framework can be specialized to encompass known QoS domains.

A Careful Handling of Monotonicity: We then study *monotonicity* in this generic QoS context, by proposing conditions enforcing it for both non-probabilistic and probabilistic QoS frameworks. Guidelines for how to enforce monotonicity are derived and ways are proposed to circumvent a lack of monotonicity. The mathematical justification of the extension required to deal with probabilistic QoS domains that are only partially, not totally ordered is non-trivial.

Support for Separation of Concerns: QoS-aware management of composite services requires developing a QoS-aware model of a service orchestration, which can be cumbersome. It is thus desirable to offer means to develop function and QoS in most possible orthogonal ways. We have developed an implementation of our mathematical approach in which QoS-aware orchestration models are automatically generated, from a specification of the function only,

[2] A dioid is a semi-ring with idempotent addition.

augmented with the declaration of the QoS domains and their algebra. This model can be executed to analyze the orchestration and perform QoS contract composition. We have implemented this technique on top of the Orc language for orchestrations [32, 35].[3]

Managing QoS by Contracts: By building on top of monotonicity, we advocate the use of *contract-based* QoS-aware management of composite services, in which the considered orchestration establishes QoS contracts with both its users and its requested services. Contract-based design amounts to performing *QoS contract composition* [40], which is the activity of estimating the tightest end-to-end QoS contract an orchestration can offer to its customer, from knowing the contract with each requested service. QoS composition is developed in Section 3.2. *Late service selection or binding* is performed on the basis of run-time QoS observations, by simply selecting, among different candidates, the one offering best QoS. Monotonicity ensures that this greedy policy will not lead to a loss in overall QoS performance of the orchestration. Best service binding is a built-in mechanism in our model, see Procedure 1 in Section 3.2. To ensure satisfaction of the QoS contract with its users, it is enough to *monitor* the conformance of each requested service with respect to its contract, since a requested service improving its QoS can only improve the overall QoS of the orchestration. This was developed in [40] for the case of latency and the techniques developed in this reference extend to multi-dimensional QoS. To account for uncertainty in QoS, *soft probabilistic QoS contracts* were proposed in [39, 40] for the case of latency and are extended in this paper to multi-dimensional QoS. Such contracts consist of the specification of a probability distribution for the QoS dimensions. Performing this requires formalizing what it means, for a service, to perform *better* than its contract. We rely for this on the notion of *stochastic ordering* [36, 37, 46] for random variables, a concept that is widely used in econometrics. All our results regarding monotonicity extend to the case in which ordering of QoS values is replaced by stochastic ordering. We can thus apply *statistical testing* [40] to detect at run time the violation of contracts in this context. To illustrate our approach, we use this tool in performing contract composition for the example TravelAgent2.

The paper is organized as follows. Related work is discussed in Section 6. Our QoS calculus is developed in Section 2; it provides the generic basis for QoS composition. Section 3 develops our theory of QoS for services orchestrations. Algebraic rules for QoS composition and best service binding are developed. Monotonicity is studied. Support for probabilistic QoS is presented. In Section 4 we present the implementation of our approach on top of the Orc language. Evaluation of this implementation on the TravelAgent2/3 is discussed in Section 5.

[3] http://orc.csres.utexas.edu/

2 QoS Calculus

In this section we develop our QoS calculus as a basis for QoS composition. A toy example is used to motivate our abstract algebra. Then we illustrate how this algebra can encompass concrete QoS domains. Finally the algebra itself is formalized in a way similar to [12, 16, 17, 23].

2.1 An Informal Introduction

In dealing with multi-dimensional QoS, several approaches can be taken. First, one can see QoS as only partially, not totally ordered. In this case QoS outcomes q and q' satisfy $q \leq q'$ if and only if $q(i) \leq q'(i)$ holds for all dimensions $i = 1 \ldots n$ of the QoS. Alternatively, one could prioritize dimensions and then take the lexicographic (total) order $q \leq q'$ iff there exists some i such that $q(j) = q'(j)$ for $j < i$ and $q(i) \leq q'(i)$. Finally, different dimensions could be weighted by considering $\sum_i w_i q(i)$ with its total order, where the w_i's are weights to be selected, e.g., by using AHP (Analytical Hierarchy Process) [47]. Finally, recall that dealing with uncertainty is by regarding QoS outcomes as random variables.

We use colored Petri nets to model the executions of a service orchestration. Queries are represented by tokens that circulate throughout the net and service calls are represented by transitions. To represent QoS measures and how they evolve while the query is being processed by the orchestration, we equip the tokens with a color, consisting of a pair

$$(v, q) = (\text{data, QoS value}). \tag{1}$$

Fig. 4 shows such a net. Each query is represented by a token entering the net at the top place. The marking shown figures the reception of such a query by the net: it results in the launching of three sub-queries in parallel. The first two sub-queries re-synchronize when calling t_2. The third sub-query branches toward either calling t_1' or calling t_1'' and then confluences. The processing of the query ends when the token reaches the exit place. With reference to this figure, the different operators needed to compute the evolution of QoS measures are introduced next. In the following discussion, we only consider choices governed by QoS (data-driven choices play no role in QoS evaluation).

We begin by giving the basic abstract operators for use in QoS management. The objective is to capture, via generic operators, how QoS measures get modified when calling a service (traversing a transition), when synchronizing the responses of services (figured by several tokens consumed by a same transition), or when different services compete against each other (such as t_1' and t_1'' in Fig. 4).

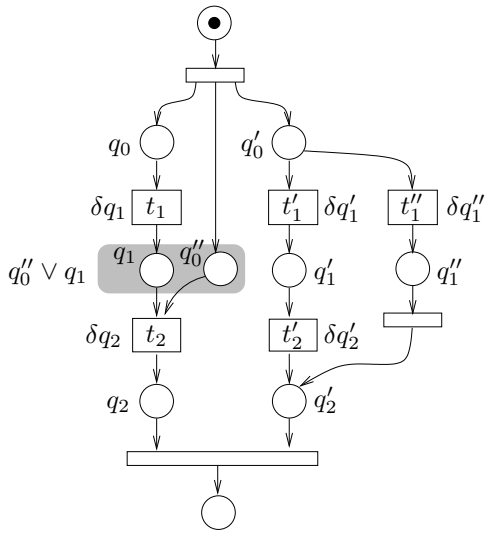

Fig. 4 A simple example. Only QoS values are mentioned — with no data. Each place comes labeled with a QoS value q which is the q-color of the token if it reaches that place.

Incrementing QoS: When traversing a transition, each token gets its QoS value incremented, which is captured by operator $\oplus$. For example, the token in the left most place has initial QoS value q_0, which gets incremented as $q_1 = q_0 \oplus \delta q_1$ when traversing transition t_1.

Synchronizing tokens: A transition t is enabled when all places in its preset have tokens. For the transition to fire, these tokens must synchronize, which results in the "worst" QoS value, denoted by the supremum $\vee$ associated to a given order $\leq$, where smaller means better. For example, when the two input tokens of t_2 get synchronized, the resulting pair of tokens has QoS $q''_0 \vee q_1$. This is depicted in Fig. 4 by the shaded area.

QoS policy: Focus on the conflict following place q'_0. The QoS alters the usual semantics of the conflict by using a *QoS policy* that is reminiscent of the classical race policy [33]. The competition between the two conflicting transitions in the post-set is solved by using order $\leq$ also used for token synchronization: test whether $q'_o \oplus \delta q'_1 \leq q'_o \oplus \delta q''_1$ holds, or the converse. The smallest of the two wins the competition—nondeterministic choice occurs if equality holds.

However, comparing $q'_o \oplus \delta q'_1$ and $q'_o \oplus \delta q''_1$ generally requires knowing the two alternatives, which in general can affect the QoS of the winner. This is taken into account by introducing a special operator "$\lhd$": If two transitions t and t' are in competition and would yield tokens with respective QoS values q and q' in their post-sets, the cost of comparing them to set the competition alters the QoS value of the winner in that—assuming the first wins—q is modified and becomes $q \lhd q'$, where $\lhd$ denotes a new operator called the

competition function. For the case of the figure, we get

$$\begin{array}{l} \text{if } (q'_o \oplus \delta q'_1) \leq (q'_o \oplus \delta q''_1) \\ \text{then } t'_1 \text{ fires and } q'_1 = (q'_o \oplus \delta q'_1) \lhd (q'_o \oplus \delta q''_1) \\ \text{if } (q'_o \oplus \delta q'_1) \geq (q'_o \oplus \delta q''_1) \\ \text{then } t''_1 \text{ fires and } q''_1 = (q'_o \oplus \delta q''_1) \lhd (q'_o \oplus \delta q'_1) \end{array} \tag{2}$$

2.2 Some Examples of QoS Domains

We now instantiate our generic framework by reviewing some examples of QoS domains, with their associated relations and operators $\oplus, \leq$, and $\lhd$.

Latency: QoS value of a token gives the accumulated latency d, or "age" of the token since it was created when querying the orchestration. Corresponding QoS domain is $\mathbb{R}_+$, equipped with $\oplus_d = +$, and $\leq_d =$ the usual order on $\mathbb{R}_+$. Regarding operator $\lhd_d$, for the case of latency with race policy [33], comparing two dates via $d_1 \leq_d d_2$ does not impact the QoS of the winner: answer to this predicate is known as soon as the first event is seen, i.e., at time $\min(d_1, d_2)$. Hence, for this case, we take $d_1 \lhd_d d_2 = d_1$, i.e., d_2 does not affect d_1. This is the basic example of QoS measure, which was studied in [15].

Security level: QoS value s of a token belongs to $(\{\text{high, low}\}, \leq_s)$, with high $\leq_s$ low. Each transition has a security level encoded in the same way, and we take $\oplus_s = \vee_s$, reflecting that a low security service processing a high security data yields a low security response. Regarding operator $\lhd_s$, again, comparing two values via $s_1 \leq_s s_2$ does not impact the QoS of the winner: QoS values are strictly "owned" by the tokens, and therefore do not interfere when comparing them. Hence, we take again $s_1 \lhd_s s_2 = s_1$, i.e., s_2 does not affect s_1. More complex partially ordered security domains can be handled similarly.

We do not claim that this solves security in orchestrations. It only serves a more modest but nevertheless useful purpose, namely to propagate and combine security levels of the requested services to derive the security level of the orchestration. How security levels of the requested services is established is a separate issue, e.g., by relying on reputation or through the negotiation of security contracts.

Reliability: Reliability is captured similarly as follows. The QoS attribute of a token takes its value in the ordered set $(\{\text{in\_operation}, \text{failed}\}, \leq_r)$, with in_operation $\leq_r$ failed. Other operators follow as for the case of Security level. By equipping this QoS domain with probability distributions we capture reliability in our setting.

Cost: QoS value c captures the total cost of building a product by assembling its parts. Referring to Fig. 4, costs are accumulated when tokens get synchronized. When a token traverses a transition, its cost is incremented according to the cost of the action being performed. A natural definition for the corresponding QoS domain would thus be $(\mathbb{D}_c, \leq_c, \oplus_c) = (\mathbb{R}, \leq, +)$ or $(\mathbb{Z}, \leq, +)$. Unfortunately, when taking this definition, synchronizing tokens using $\vee_c$ amounts to taking the worst cost, which is not what we need. We need instead the sum of the costs of incoming tokens, an operation different from $\vee_c$.

The right idea is to encode the cost by using multi-sets. The overall cost held by a token is obtained by adding the costs of the constituting parts plus the costs of successive assembly actions. Parts and actions are then handled as "quanta of cost" and the token collects them while traversing the orchestration. This leads to defining the QoS domain as a multi-set of *cost types*: $\mathbb{D}_c = \mathbf{Q} \mapsto \mathbb{N}$, where $\mathbf{Q}$ is a set of cost types equipped with a cost labeling function $\lambda : \mathbf{Q} \mapsto \mathbb{R}_+$. Each $\mathbf{q} \in \mathbf{Q}$ corresponds to either a part or an assembly action and has a unique identifier. Domain $\mathbb{D}_c$ is equipped with the partial order of functions and $\vee_c$ follows as the corresponding least upper bound. Recall that operator $\vee_c$ is used to synchronize tokens, see Fig. 4. In this context, it makes sense to assume that cost types held by the tokens for synchronization are different. For this case, $\vee_c$ coincides with the addition of multi-sets and costs get added as wished. Traversing a transition amounts to adding the corresponding quantum in the set, hence, identifying singletons with the corresponding element, $\oplus_c$ is again the addition of multi-sets. Finally, $(\mathbb{D}_c, \leq_c, \oplus_c) = (\mathbf{Q} \mapsto \mathbb{N}, \subseteq, +)$. As before, the competition function is $c_1 \lhd_c c_2 = c_1$ when $c_1 \leq_c c_2$, i.e., c_2 does not affect c_1.

Composite QoS, first example: We may also consider a composite QoS measure consisting of the pair (s, r), where s is as above and r is some *Quality of Response* with domain $\mathbb{D}_r$, equipped with $\leq_r$ and $\lhd_r$. Since the two components s and r are similar in nature, we simply take $\leq = \leq_s \times \leq_r$ and $\lhd = (\lhd_s, \lhd_r)$.

Composite QoS, second example: So far the special operator $\lhd$ did not play any role. We will need it, however, for the coming case, in which we consider a composite QoS measure (s, d), where s and d are as above. We want to give priority to security s, and thus we now take $\leq$ to be the lexicographic order obtained from the pair $(\leq_s, \leq_d)$ by giving priority to s.

Focus on operator $\lhd$. Consider the marking resulting after firing t_1 and t_1' in Fig. 4, enabling t_2 and t_2', which are in conflict. Let the QoS value of the token in postset of t_2, i.e. $q_2 = (low, d_2)$. (Recall that $q_2 = (q_o'' \vee q_1) \oplus \delta q_2$.) Similarly, let $q_2' = (low, d_2')$ where $d_2' >_d d_2$. From the competition rule, transition t_2 wins the conflict and the outgoing token has QoS value $q_2 = (low, d_2)$. However, the decision to select t_2 can only be made when q_2' is known, that is, at time d_2'. The reason for this is that, since at time d_2 a token with security level *low* is seen at place following t_2, it might be that a token with security level *high* later enters place following t_2'. The latter would win the conflict according to our QoS policy — security level prevails. Observing that the right most

token indeed has priority level *low* can only be seen at time d'_2. Thus it makes little sense assigning $q_2 = (low, d_2)$ to the outgoing token; it should rather be $q_2 = (low, d'_2)$. This is why a non-trivial operator $\lhd$ is needed, namely, writing $\leq$ for short instead of $\leq_d$:

$$\begin{array}{l}(s,d) \lhd (s',d') \;=\\ \textbf{if } d \leq d' \;\text{ and } s = low \textbf{ then}(s,d') \textbf{ else } (s,d)\end{array} \tag{3}$$

2.3 The QoS Calculus

In this section we formalize the discussion of Section 2.1. We introduce algebraic QoS domains. Our framework is a mild modification of the one proposed by [12, 16, 17, 23], based on *semi-rings*. Besides some minor adaptations, the main difference lies in the consideration of the "competition function".

Definition 1 (QoS domain) *A* QoS domain *is a tuple* $\mathbb{Q} = (\mathbb{D}, \leq, \oplus, \lhd)$ *where:*

- $(\mathbb{D}, \leq)$ *is a partial order that is a complete upper lattice, meaning that every subset* $S \subseteq \mathbb{D}$ *has a least upper bound denoted by* $\bigvee S$*. For any* $S \subseteq \mathbb{D}$, $\min(S)$ *denotes the set of all* $q \in S$ *such that no* $q' \in S$ *exists such that* $q' < q$*, with a symmetric definition for* $\max(S)$.
- *Operator* $\oplus : \mathbb{D} \times \mathbb{D} \to \mathbb{D}$ *is a commutative semi-group with neutral element* 0 *and such that:*

$$\textit{monotonicity: } \left.\begin{array}{l} q_1 \leq q'_1 \\ q_2 \leq q'_2 \end{array}\right\} \implies (q_1 \oplus q_2) \leq (q'_1 \oplus q'_2) \tag{4}$$

$$\forall q, q' \in \mathbb{D}, \exists q'' \in \mathbb{D} \implies q \leq q' \oplus q'' \tag{5}$$

- *The* competition function $\lhd : \mathbb{D} \times 2^{\mathbb{D}} \to \mathbb{D}$ *satisfies:*

$$q \lhd \epsilon = q \textit{ where } \epsilon \textit{ denotes the empty set} \tag{6}$$

$$\left.\begin{array}{l} q \leq q' \\ q_1 \leq q'_1 \\ \vdots \\ q_n \leq q'_n \end{array}\right\} \implies q \lhd \{q_1 \ldots q_n\} \leq q' \lhd \{q'_1 \ldots q'_n\} \tag{7}$$

$$q_0 < q_1 \implies q_0 \lhd \{q_1, q_2 \ldots q_n\} \leq q_1 \lhd \{q_0, q_2 \ldots q_n\} \tag{8}$$

$$\forall i = 1 \ldots n : q_i \leq q \implies q \lhd \{q_1 \ldots q_n\} = q \tag{9}$$

Referring to our motivating discussion: $\mathbb{D}$ is the set in which QoS takes its values; $q \leq q'$ is interpreted as "q is better than (or preferred to) q'"; partial order $\leq$ gives raise to the least upper bound $\vee$, interpreted as the worst QoS; operator $\oplus$ is used to accumulate QoS quanta from causally related events; its condition (5) will play an important role in the study of monotonicity. The competition function $\lhd$ accounts for the additional cost of comparing the QoS of competing events, additional cost induced on the winning event.

The special monotonicity conditions (8) and (9) for the competition function ensure that taking into account the cost of comparing will not revert the QoS-based ordering of the events under comparison. The actual size of the second component of $\lhd$ depends on the considered event, this is why the domain of $\lhd$ is $\mathbb{D} \times 2^{\mathbb{D}}$. Examples were given in Section 2.1. It is easily checked that axioms are met by these examples.

If some QoS measure q of the orchestration is irrelevant to a service it involves, we take the convention that this service acts on tokens with a 0 increment on the value of q. With this convention we can safely assume that the orchestration, all its requested services, and all its tokens use the same QoS domain. This assumption will be in force in the sequel.

3 A QoS framework for composite services

This section collects the technical material in support of our theory and developments. We first recall the needed background on Petri nets as a supporting framework for service orchestrations—to simplify our presentation we restrict ourselves to safe free choice nets, see below. On top of this framework, we define priority rules for QoS based selection of competing services and we develop OrchNets as a model of QoS-sensitive composite services. We then study monotonicity. The above material is subsequently lifted to probabilistic QoS. We conclude by some methodological discussion.

3.1 Petri Nets, Occurrence Nets, Orchestration Nets

A *Petri net* [38] is a tuple $N = (\mathcal{P}, \mathcal{T}, \mathcal{F}, M_0)$, where: $\mathcal{P}$ is a set of *places*, $\mathcal{T}$ is a set of *transitions* such that $\mathcal{P} \cap \mathcal{T} = \emptyset$, $\mathcal{F} \subseteq (\mathcal{P} \times \mathcal{T}) \cup (\mathcal{T} \times \mathcal{P})$ is the *flow relation*. For $x \in \mathcal{P} \cup \mathcal{T}$, we call ${}^\bullet x = \{y \mid (y, x) \in \mathcal{F}\}$ the *preset* of x and $x^\bullet = \{y \mid (x, y) \in \mathcal{F}\}$ the *postset* of x. A *marking* is a map $M : \mathcal{P} \to \mathbf{N}$; in the tuple defining N, M_0 is the *initial marking*. *Firing* transition t at marking M requires $M(p) > 0$ for every $p \in {}^\bullet t$ and yields the new marking M' such that $M'(p) = M(p) - 1$ for $p \in {}^\bullet t \setminus t^\bullet$, $M'(p) = M(p) + 1$ for $p \in t^\bullet \setminus {}^\bullet t$, and $M'(p) = M(p)$ otherwise.

For a net $N = (\mathcal{P}, \mathcal{T}, \mathcal{F}, \mathcal{R}, M_0)$ the *causality relation* $\preceq$ is the transitive and reflexive closure of $\mathcal{F}$ and we set $\prec = \preceq \cap \neq$. For a node $x \in \mathcal{P} \cup \mathcal{T}$, the set of *causes* of x is $\lceil x \rceil = \{y \in \mathcal{P} \cup \mathcal{T} \mid y \preceq x\}$. Say that two transitions t, t' are in *conflict*, written $t \# t'$, if ${}^\bullet t \cap {}^\bullet t' \neq \emptyset$ or t and t' possess some causes that are in conflict. Say that net N is *free choice* if the relation $\{(t, t') \mid {}^\bullet t \cap {}^\bullet t' \neq \emptyset\}$ forms a partition of $\mathcal{T}$. If N is free choice, a *cluster* [38] is a minimal set $\mathbf{c}$ of places and transitions of N such that

$$\begin{array}{l} \forall t \in \mathbf{c} \implies {}^\bullet t \subseteq \mathbf{c} \\ \forall p \in \mathbf{c} \implies p^\bullet \subseteq \mathbf{c} \end{array} \tag{10}$$

Any two distinct transitions of a same cluster are in conflict and clusters form a partition of the set of all nodes of a free choice net.

Occurrence nets: A Petri net is *safe* if all its reachable markings M satisfy $M(\mathcal{P}) \subseteq \{0,1\}$. A safe net $N = (\mathcal{P}, \mathcal{T}, \mathcal{F}, M_0)$ is an *occurrence net (O-net)* iff

1. $\preceq$ is a partial order and $\lceil t \rceil$ is finite for any $t \in \mathcal{T}$;
2. for each place $p \in \mathcal{P}$, $|{}^\bullet p| \leq 1$;
3. for each $t \in \mathcal{T}$, $\neg \#\lceil t \rceil$ holds;
4. $M_0 = \{p \in \mathcal{P} | {}^\bullet p = \emptyset\}$ holds.

A *configuration* of N is a subnet κ of nodes of N such that: 1) κ is *causally closed*, *i.e*, if $x \preceq x'$ and $x' \in \kappa$ then $x \in \kappa$; and, 2) κ is *conflict-free*. For convenience, we require that the maximal nodes in a configuration are places. A configuration κ_2 is said to extend configuration κ_1 (written as $\kappa_1 \preceq \kappa_2$) if $\kappa_1 \subseteq \kappa_2$ and $\nexists t \in \kappa_2 \setminus \kappa_1, t' \in \kappa_1$ such that $t \nearrow t'$. Two configurations κ and κ' are said to be *compatible* if 1) $\kappa \cup \kappa'$ is a configuration, and 2) $\kappa \preceq \kappa \cup \kappa'$ and $\kappa' \preceq \kappa \cup \kappa'$. Node x is called compatible with configuration κ if $\lceil x \rceil$ and κ are compatible. Transition t is *enabled* by κ if $t \notin \kappa$ and $\kappa \cup \{t\} \cup t^\bullet$ is a configuration. For κ a configuration, its *future* N^κ is defined as

$$\begin{array}{l} N^\kappa = \mathit{maxPlaces}(\kappa) \cup \\ \{x \in \mathcal{P} \cup \mathcal{T} \mid x \notin \kappa \text{ and } x \text{ is compatible with } \kappa\} \end{array} \tag{11}$$

where $\mathit{maxPlaces}(\kappa)$ is the set of maximal nodes of κ (which are all places). Two nodes x and y are said to be *concurrent* if they are neither in conflict nor causally related.

Unfoldings and Orchestration nets: The executions of a safe Petri net N can be represented by its *unfolding* U_N, which is an occurrence net collecting all executions of N in such a way that common prefixes are represented once. For example, Fig. 4 shows a net, the unfolding of which is obtained by removing the maximal (exit) place and attaching a different copy of this exit place to each exit transition. Formally, unfolding U_N is derived from N [24] in the following way. For $N = (\mathcal{P}, \mathcal{T}, \mathcal{F}, \mathcal{R}, M_0)$ and $N' = (\mathcal{P}', \mathcal{T}', \mathcal{F}', \mathcal{R}', M_0')$ two safe Petri nets, a *morphism* $\varphi : N \to N'$ is a function from $\mathcal{P} \cup \mathcal{T}$ to $\mathcal{P}' \cup \mathcal{T}'$, mapping $\mathcal{P}$ to $\mathcal{P}'$ and $\mathcal{T}$ to $\mathcal{T}'$, preserving the initial marking: $\varphi(M_0) = M_0'$, and preserving the flow and read relations: $\varphi({}^\bullet t) = {}^\bullet \varphi(t)$, $\varphi(t^\bullet) = \varphi(t)^\bullet$, and $\varphi({}^\circ t) = {}^\circ \varphi(t)$. If N' is another occurrence net and $\psi' : N' \to N$ is a morphism, then there exists a third morphism $\psi : N' \to U_N$ such that ψ' factorizes as $\psi' = \varphi \circ \psi$, where $\circ$ is the composition of functions. This property characterizes the unfolding U_N. If net N is free choice, then so is its unfolding U_N.

Definition 2 (orchestration net) *Call* Orchestration net *any free choice safe Petri net possessing a finite unfolding.*

We insist that Petri nets with loops can still possess a finite unfolding. An example of this is the Petri net modeling the examples TravelAgent of Fig. 2 and Fig. 3, which involve successive retries guarded by a timeout. In the sequel we only consider Petri nets that are orchestration nets. Examples of Orchestration nets are the loop-free and 1-safe WorkFlow nets (WFnets). WF-nets were proposed by van der Aalst [2,4,5] and are Petri nets with a special initial

place (where the initial tokens are provided) and a special final place (from which tokens exit the net).

3.2 OrchNets

The OrchNets we propose as a model to capture QoS in composite services are a special form of *colored occurrence nets (CO-nets)*. Executions of Workflow Nets [2,3] are also CO-nets. The reader can compare our approach with the graph-based approach of [48].

Throughout this section we assume a QoS domain $(\mathbb{D}, \leq, \oplus, \lhd)$. OrchNets formalize the notion of an orchestration with its QoS. The mathematical semantics of OrchNets formalizes QoS contract composition, i.e., the process of deriving end-to-end QoS of the orchestration from the QoS of its involved services.

Definition 3 (OrchNet) *An* OrchNet *is a tuple* $\mathcal{N} = (N, V, Q, Q_{\text{init}})$ *consisting of*

- *A finite free choice occurrence net* N *with token attributes*

$$c = (v, q) = (\text{data, QoS value})$$

- *A family* $V = (\nu_t)_{t \in \mathcal{T}}$ *of* value functions, *mapping the data values of the transition's input tokens to the data value of the transition's output token.*
- *A family* $Q = (\xi_t)_{t \in \mathcal{T}}$ *of* QoS functions, *mapping the data values of the transition's input tokens to a QoS increment.*
- *A family* $Q_{\text{init}} = (\xi_p)_{p \in \min(\mathcal{P})}$ *of* initial QoS functions *for the minimal places of* N.

Value and QoS functions can be nondeterministic.

The nondeterminism of a function can be resolved by introducing an explicit daemon ω making choices explicit. As a result, $\nu_t(\omega), \xi_t(\omega)$, and $\xi_p(\omega)$ are all deterministic functions of their respective inputs. We denote by Ω the set of all daemons.[4]

We now explain how the presence of QoS values attached to tokens affects the semantics of OrchNets. Any place p of occurrence net N has a pair $(v_p, q_p) =$ (data, QoS value) assigned to it, which is the color held by a token reaching that place. In the following QoS policy, the role of data in the semantics has been abstracted—taking it into account would only increase the notational burden without introducing changes worth the study.

Procedure 1 (QoS aware semantics) *Let* $\omega \in \Omega$ *be any value for the daemon. The continuation of any finite configuration* $\kappa(\omega)$ *is constructed by performing the following steps, where we omit the explicit dependency of* $\kappa(\omega)$, $\nu_t(\omega)$, *and* $\xi_t(\omega)$, *with respect to* ω*:*

[4] The *schedulers* introduced for probabilistic automata by Lynch and Segala [45] are a special case of daemon.

1. *Choose nondeterministically a $\preceq$-minimal cluster* $\mathbf{c}$ *in the future of* κ.
2. *For every* $t \in \mathbf{c}$, *compute:*

$$q_t = \left(\bigvee_{p' \in {}^\bullet t} q_{p'}\right) \oplus \xi_t(v_{p'} \mid p' \in {}^\bullet t) \tag{12}$$

3. *Competition step: select nondeterministically a minimal transition* t_* *of* $\mathbf{c}$ *such that no other minimal transition* t *of* $\mathbf{c}$ *exists with* $q_t < q_{t_*}$. *The set* Ω *of daemons is extended to resolve this additional nondeterminism.*
4. *Augment* κ *to* $\kappa' = \kappa \cup \{t_*\} \cup t_*^\bullet$, *and assign, to every* $p \in t_*^\bullet$, *the pair* (v, q), *where*

$$\begin{array}{l} v = \nu_t(v_{p'} \mid p' \in {}^\bullet t) \\ q = q_{t_*} \lhd \{q_t \mid t \in \mathbf{c}, t \neq t_*\} \end{array} \tag{13}$$

Competition step 3 formalizes on-line service binding based on best QoS. Step 4 of QoS policy simplifies for all examples of Section 2.1 by not needing the second formula of (13), except for the last one, see formula (3). Observe that the augmented configuration κ' as well as the pair (v, q) depend on ω. We are now ready to formalize what the set Ω of all daemons should be for Procedure 1.

Defining the set Ω *of all daemons:* The nondeterminism of a function mapping X to Y can be resolved by introducing an explicit daemon making choices explicit. For X and Y two sets, call (X, Y)-*daemon* (or simply *daemon* if no confusion can result) any total function

$$\omega : X \times 2^Y \to Y \tag{14}$$

The set of all (X, Y)-daemons is denoted by Ω_{XY} or simply Ω. Determinizing a nondeterministic function $\chi : X \to 2^Y$ consists in selecting a daemon $\omega \in \Omega$, which fixes the (deterministic) function

$$\chi^\omega(x) =_{\text{def}} \omega(x, \chi(x)).$$

This construction is implicitly invoked each time a daemon is mentioned. To explicit what the set Ω of all daemons should be, for Procedure 1, we first identify the different sources of nondeterminism arising in this procedure. First of all, the nondeterminism in the choice of the minimal cluster $\mathbf{c}$ in Step 1 does not need to get resolved since it yields a confluent evaluation of the end-to-end QoS of configurations, because all minimal clusters are concurrent and $\oplus$ is commutative and associative. Consequently, sources of nondeterminism for consideration are 1) the ν_t and ξ_t for every $t \in \mathcal{T}$ (the set of transitions of N), and 2) the nondeterministic selection of the optimal transition in Step 3. Denoting by $\mathbf{C}$ the set of all clusters of N, we apply construction (14) with

$$\begin{array}{ll} X' = \mathcal{T} \text{ and } Y' = D \times \mathbb{D} & \text{which yields } \Omega' \\ X'' = \mathbf{C} \text{ and } Y'' = \mathcal{T} & \text{which yields } \Omega'' \end{array}$$

where D is the domain of data, and set

$$\Omega =_{\text{def}} \Omega' \times \Omega'' \tag{15}$$

Component ω' resolves the nondeterminism of ν_t and ξ_t, whereas component ω'' resolves the nondeterminism in selecting the optimal transition in Step 3.

Since occurrence net N is finite, the QoS policy terminates in finitely many steps when $N^{\kappa(\omega)} = \emptyset$. The total execution thus proceeds by a finite chain of nested configurations: $\emptyset = \kappa_0(\omega) \prec \kappa_1(\omega) \ldots \prec \kappa_n(\omega)$. Hence, $\kappa_n(\omega)$ is a maximal configuration of $\mathcal{N}$ that can actually occur according to the QoS policy, for a given $\omega \in \Omega$. We generically denote this maximal configuration by

$$\kappa(\mathcal{N}, \omega). \tag{16}$$

For the example of latency, our QoS policy boils down to the classical *race policy* [33]. In general, our QoS policy bears some similarity with the "pre-selection policies" of [33], except that the continuation is selected based on QoS values in our case, not on random selection. We will also need to compute the QoS for *any* configuration of N, even if it is not a winner of the competition policy. We do this by modifying Procedure 1 as follows:

Procedure 2 (QoS of an arbitrary configuration) *Let $\kappa_{\max}$ be any maximal configuration of N and $\kappa \preceq \kappa_{\max}$ a prefix of it. With reference to Procedure 1, perform: step 1 with* **c** *any $\preceq$-minimal cluster in $\kappa_{\max} \setminus \kappa$, step 2 with no change, and then step 4 for any t as in step 2. Performing this repeatedly yields the pair (v_p, q_p) for each place p of $\kappa_{\max}$.* □

We are now ready to define what the QoS value of an OrchNet is:

Definition 4 (End-to-end QoS) *For κ any configuration of occurrence net N, and ω any value for the daemon, the* end-to-end QoS *of κ is defined as*

$$E_\omega(\kappa, \mathcal{N}) = \bigvee_{p \in maxPlaces(\kappa)} q_p(\omega) \tag{17}$$

The end-to-end QoS $E_\omega(\mathcal{N})$ *and* pessimistic end-to-end QoS $F_\omega(\mathcal{N})$ *of OrchNet $\mathcal{N}$ are respectively given by*

$$E_\omega(\mathcal{N}) = E_\omega(\kappa(\mathcal{N}, \omega), \mathcal{N}) \tag{18}$$

$$F_\omega(\mathcal{N}) = \max\{E_\omega(\kappa, \mathcal{N}) \mid \kappa \in \mathcal{V}(N)\} \tag{19}$$

where function max *picks one of the maximal values in a partially ordered set, $\kappa(\mathcal{N}, \omega)$ is defined in (16), and $\mathcal{V}(N)$ is the set of all maximal configurations of net N.*

Observe that $E_\omega(\mathcal{N}) \leq F_\omega(\mathcal{N})$ holds and $E_\omega(\mathcal{N})$ is indeed observed when the orchestration is executed. The reason for considering in addition $F_\omega(\mathcal{N})$ will be made clear in the next section on monotonicity.

So far formulas (18) and (19) provide the composition rules for deriving the end-to-end QoS for each individual call to the orchestration. Monte-Carlo simulation techniques can then be used on top of (18) and (19) to derive the end-to-end probabilistic QoS contract from the contracts negotiated with the requested services [39, 40]. See also [28] for fast Monte-Carlo simulation techniques.

3.3 Monotonicity

The monotonicity of an orchestration with respect to QoS is studied in this section, for the non-probabilistic setting. Extension to the probabilistic setting is discussed in Section 3.4. We provide sufficient and structurally necessary conditions for monotonicity, when QoS is measured in terms of tight end-to-end QoS—missing proofs are deferred to Appendix A. When these conditions fail to hold, then pessimistic end-to-end QoS can be considered when dealing with contracts, as monotonicity is always guaranteed when using it. Monotonicity is assumed in the rest of the paper. Also, to simplify the presentation, the following assumption will be in force:

Assumption 1 *QoS functions ξ_t can be increased at will within their respective domain of values, independently for each transition t.*

This is only a technical assumption. This assumption rules out cases in which one requires, e.g., that QoS functions ξ_t and $\xi_{t'}$ can be modified at will, but subject to the constraint $\xi_t = \xi_{t'}$. The general case yields the same results, at the price of more complex notations. The reader interested in the general case is referred to [41].

For two families Q and Q' of QoS functions, write $Q' \geq Q$ and $Q'_{\text{init}} \geq Q_{\text{init}}$ to mean:

$$\begin{array}{c} \forall \omega \in \Omega, \forall t \in \mathcal{T} \Rightarrow \xi'_t(\omega) \geq \xi_t(\omega) \\ \text{respectively } \forall t \in \mathcal{T} \Rightarrow Q_{\text{init}}(t) \geq Q_{\text{init}}(t) \end{array} \tag{20}$$

For $\mathcal{N}' = (N, V, Q', Q'_{\text{init}})$ (observe that N and V are unchanged), write

$$(i) : \mathcal{N}' \geq \mathcal{N}; \quad (ii) : E(\mathcal{N}') \geq E(\mathcal{N}); \quad (iii) : F(\mathcal{N}') \geq F(\mathcal{N})$$

to mean, respectively:

(i): $Q' \geq Q$ and $Q'_{\text{init}} \geq Q_{\text{init}}$ both hold;
(ii): $\forall \omega \in \Omega$, $E_\omega(\mathcal{N}') \geq E_\omega(\mathcal{N})$ holds;
(iii): $\forall \omega \in \Omega$, $F_\omega(\mathcal{N}') \geq F_\omega(\mathcal{N})$ holds.

Definition 5 *Call OrchNet $\mathcal{N}$* monotonic *if*

$$\forall \mathcal{N}' : \mathcal{N}' \geq \mathcal{N} \implies E(\mathcal{N}') \geq E(\mathcal{N})$$

Call OrchNet $\mathcal{N}$ pessimistically monotonic *if*

$$\forall \mathcal{N}' : \mathcal{N}' \geq \mathcal{N} \implies F(\mathcal{N}') \geq F(\mathcal{N})$$

The following immediate result justifies considering also the pessimistic end-to-end QoS:

Theorem 1 *Any OrchNet is pessimistically monotonic.*

Consequently, it is always sound to base contract composition and contract monitoring [40] on pessimistic end-to-end QoS. This, however, has a price, since pessimistic end-to-end QoS is pessimistic compared to (actual) end-to-end QoS. The next theorem gives conditions enforcing monotonicity:

Theorem 2 *OrchNet* $\mathcal{N} = (N, V, Q, Q_{\text{init}})$ *is monotonic if and only if:*

$$\forall \omega \in \Omega, \forall \kappa \in \mathcal{V}(N) \implies E_\omega(\kappa, \mathcal{N}) \geq E_\omega(\kappa(\mathcal{N}, \omega), \mathcal{N}) \tag{21}$$

where $\mathcal{V}(N)$ *is the set of all maximal configurations of net* N *and* $\kappa(\mathcal{N}, \omega)$ *is defined in* (16).

Condition (21) expresses that Procedure 1 implements globally optimal service selection. It is costly to verify and may not even be decidable in general.

Thus, we develop a structural condition for monotonicity for *Orchestration nets* N (Definition 2). Orchestration net N induces an OrchNet $\mathcal{N}_N = (U_N, \nu_N, Q_N, Q_{\text{init}})$ by attaching, to each transition t of the unfolding U_N of N, the value and QoS inherited from N through the unfolding $N \mapsto U_N$.

Theorem 3 *A sufficient condition for the OrchNet* $\mathcal{N}_N = (U_N, \nu_N, Q_N, Q_{\text{init}})$ *to be monotonic is that every cluster* $\mathbf{c}$ *of* N *satisfies the following condition:*

$$\forall t_1, t_2 \in \mathbf{c}, t_1 \neq t_2 \implies t_1^\bullet = t_2^\bullet. \tag{22}$$

If, in addition, every transition of N *is reachable and partial order* $(\mathbb{D}, \leq)$ *is such that for every* $q \in \mathbb{D}$*, there exists* $q' \in \mathbb{D}$ *such that* $q' > q$*, then* (22) *is also necessary.*

In words, a sufficient condition for monotonicity is that, each time branching has occurred in net N, a join occurs right after. The additional condition ensuring necessity is a reinforcement of condition (5).

3.4 Probabilistic Monotonicity

To account for uncertainties in QoS performance, soft probabilistic contracts were proposed in [39], with associated composition and monitoring procedures, for the particular case of response time. In [40,42] the above approach was extended to more general QoS. In this section, we describe the corresponding model of *probabilistic OrchNets*, an extension of OrchNets supporting probabilistic behavior of QoS measures. Details are found in [41].

In probabilistic OrchNets, the nondeterministic QoS functions ξ_t are now random, and so are the non-deterministic selections of minima in competition step of Procedure 1. Equivalently, the set Ω for the values of the daemon is equipped with some probability $\mathbf{P}$. To define monotonicity, we need to give a meaning to (20) when ξ_t is random. This is achieved by considering the *stochastic partial order* [46] induced by partial order $\leq$ defined on $\mathbb{D}$. We briefly recall this notion next. Consider *ideals* of $\mathbb{D}$, i.e., subsets I of $\mathbb{D}$ that are downward closed: $x \in I$ and $y \leq x \implies y \in I$. Examples of ideals are:

for $\mathbb{R}_+$, the intervals, $[0,x]$ for all x; for $\mathbb{R}_+ \times \mathbb{R}_+$ equipped with the product order, arbitrary unions of rectangles $[0,x] \times [0,y]$. Now, if ξ has values in $\mathbb{D}$, we define its *distribution function* by $F(I) = \mathbf{P}(\xi \in I)$, for I ranging over the set of all ideals of $\mathbb{D}$. For ξ and ξ' two random variables with values in $\mathbb{D}$, with respective distribution functions F and F', define

$$\xi \geq^s \xi' \text{ iff for any ideal } I \text{ of } \mathbb{D},\ F(I) \leq F'(I) \text{ holds.} \tag{23}$$

With this new interpretation of the order, we will now show that Theorems 1–3 remain valid. We first define probabilistic OrchNets, which are OrchNets in which the QoS of the different services are randomized.

Definition 6 (probabilistic OrchNet) *A* probabilistic OrchNet *is a pair* $(\mathcal{N}, \mathbf{P})$ *consisting of an OrchNet* $\mathcal{N}$ *following Definition 3 and a probability distribution* $\mathbf{P}$ *over the set* Ω *of daemons of* $\mathcal{N}$ *equipped with its Borel* σ*-algebra.*

We further assume that the random variables $\nu_t(\omega), \xi_t(\omega)$*, where* t *ranges over the set* $\mathcal{T}$ *of all transitions of the OrchNet, and the different random selections of an optimum in Step 3 of Procedure 1 are all mutually independent.*

How can we lift monotonicity to this probabilistic setting? We first make precise what the set Ω of all daemons is. For t a generic transition, let $(\Omega_t, \mathbf{P}_t)$ be the set of possible experiments together with associated probability, for random latency ξ_t; and similarly for $(\Omega_{\mathbf{c}}, \mathbf{P}_{\mathbf{c}})$, where $\mathbf{c}$ ranges over the set $\mathbf{C}$ of all clusters of N. Thanks to the assumption stated at the end of Definition 6, setting

$$\Omega = \Big(\prod_{t\in\mathcal{T}} \Omega_t\Big) \times \Big(\prod_{\mathbf{c}\in\mathbf{C}} \Omega_{\mathbf{c}}\Big) \text{ and } \mathbf{P} = \Big(\prod_{t\in\mathcal{T}} \mathbf{P}_t\Big) \times \Big(\prod_{\mathbf{c}\in\mathbf{C}} \mathbf{P}_{\mathbf{c}}\Big) \tag{24}$$

yields the probabilistic part of Definition 6. In the nondeterministic framework of Section 3.3, we said that

$$\xi \geq \xi' \text{ if } \xi(\omega) \geq \xi'(\omega) \text{ holds } \forall \omega \in \Omega \tag{25}$$

Clearly, if two random latencies ξ and ξ' satisfy condition (25), then they also satisfy condition (23). That is, *ordering* (25) *is stronger than stochastic ordering* (23). Unfortunately, the converse is *not* true in general. For example, condition (23) may hold while ξ and ξ' are two independent random variables, which prevents (25) from being satisfied. Nonetheless, the following result holds [46], which will allow us to proceed:

Theorem 4 *Assume condition* (23) *holds for the two distribution functions* F *and* F'*. Then, there exists a probability space* Ω*, a probability* $\mathbf{P}$ *over* Ω*, and two real valued random variables* $\hat{\xi}$ *and* $\hat{\xi}'$ *over* Ω*, such that:*

1. $\hat{\xi}$ *and* $\hat{\xi}'$ *possess* F *and* F' *as respective distribution functions, and*
2. *condition* (25) *is satisfied by the pair* $(\hat{\xi}, \hat{\xi}')$ *with probability* 1.

The proof of this result is immediate if $(\mathbb{D}, \leq)$ is a total order. It is, however, highly nontrivial if $\leq$ is only a partial order. This theorem is indeed part of theorem 1 of [46].[5] Theorem 4 allows to reduce the stochastic comparison

[5] Thanks are due to Bernard Delyon who pointed this reference to us.

of random variables to their ordinary comparison as functions defined over the same set of experiments endowed with a same probability. This applies in particular to each random QoS function and each random initial QoS function, when considered in isolation. Thus, when performing construction (24) for two OrchNets $\mathcal{N}$ and $\mathcal{N}'$, we can take the *same* pair $(\Omega_t, \mathbf{P}_t)$ to represent both ξ_t and ξ'_t, and similarly for ξ_p and ξ'_p. Applying (24) implies that both $\mathcal{N}$ and $\mathcal{N}'$ are represented using the *same* pair $(\Omega, \mathbf{P})$. This leads naturally to Definition 6.

In addition, applying Theorem 4 to each transition t and each minimal place p yields that stochastic ordering $\mathcal{N} \geq^s \mathcal{N}'$ reduces to ordinary ordering $\mathcal{N} \geq \mathcal{N}'$. Observe that this trick does not apply to the overall QoS $E(\mathcal{N})$ and $E(\mathcal{N}')$ of the two OrchNets; the reason for this is that the space of experiments for these two random variables is already fixed (it is Ω) and cannot further be played with as theorem 4 requires. Thus we can reformulate probabilistic monotonicity as follows—compare with Definition 5:

Definition 7 *Probabilistic OrchNet* $(\mathcal{N}, \mathbf{P})$ *is called* probabilistically monotonic *if, for any probabilistic OrchNet* $\mathcal{N}'$ *such that* $\mathcal{N} \geq \mathcal{N}'$*, we have* $E(\mathcal{N}) \geq^s E(\mathcal{N}')$.

Note the careful use of $\geq$ and $\geq^s$. The following two results establish a relation between probabilistic monotonicity and monotonicity.

Theorem 5 *If OrchNet* $\mathcal{N}$ *is monotonic, then, probabilistic OrchNet* $(\mathcal{N}, \mathbf{P})$ *is probabilistically monotonic for any probability* $\mathbf{P}$ *over the set* Ω*. Vice-versa, if probabilistic OrchNet* $(\mathcal{N}, \mathbf{P})$ *is probabilistically monotonic, then* $\mathcal{N}$ *is monotonic with* $\mathbf{P}$*-probability* 1.

As a consequence, Theorem 3 enforcing monotonicity extends to the probabilistic setting.

3.5 Enforcing Monotonicity

Theorem 3 in Section 3.3 provides guidelines regarding how to enforce monotonicity. Consider again the workflow of Fig. 4 and the two alternative branches beginning at the place labeled with QoS q'_0 and ending at the place labeled with the QoS q'_2. This pattern is a source of non-monotonicity as we have seen. One way of enforcing monotonicity is by invoking Theorem 3. Aggregate the two successive transitions in each branch and regard the result as a single transition (t'_{12} for the left branch and t''_{12} for the right branch). The QoS increments of t'_{12} and t''_{12} are equal to $\delta q'_{12} = \delta q'_1 \oplus \delta q'_2$ and $\delta q''_{12} = \delta q''_1 \oplus \delta q''_2$, respectively. The resulting Orchestration net satisfies the condition of Theorem 3 and thus is monotonic. This process of aggregation is illustrated on Fig. 5, mid diagram.

An alternative to the above procedure consists in not modifying the orchestration but rather changing the QoS evaluation procedure. Referring again to

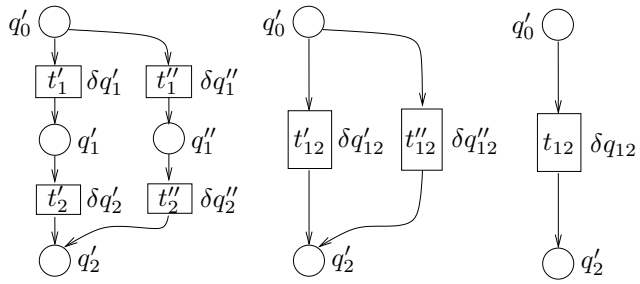

Fig. 5 Enforcing monotonicity through service aggregation, mid diagram, with $\delta q'_{12} = \delta q'_1 \oplus \delta q'_2$ and $\delta q''_{12} = \delta q''_1 \oplus \delta q''_2$. Pessimistic QoS evaluation, right diagram, with $\delta q_{12} = \delta q'_{12} \vee \delta q''_{12}$.

Fig. 4, isolate the part of the workflow that is a source of non-monotonicity, namely the subnet shown on Fig. 5, left. For this subnet, use pessimistic formula (19) to get a pessimistic but monotonic bound for the QoS of this subnet. For this example, the pessimistic bound is equal to $\delta q_{12} = \delta q'_{12} \vee \delta q''_{12}$. We then plug the result in the evaluation of the QoS of the overall orchestration, by aggregating the isolated subnet into a single transition t_{12}, with QoS increment δq_{12}. This is illustrated on Fig. 5, right diagram.

The above two procedures yield different results. By aggregating service calls performed in sequence, the first procedure delays the selection of the best branch. The second procedure does not suffer from this drawback. In turn, it results in a pessimistic evaluation of the end-to-end QoS. Both approaches restore monotonicity.

4 Implementing our approach in Orc

We have implemented our approach on top of the Orc orchestration language and we now present two aspects of this implementation: we explain how our approach supports separation of concerns in QoS-aware orchestration modeling; we also illustrate contract composition as a method for QoS-based design of composite services. Before presenting this, we briefly summarize how the technical developments of Section 3 contribute to our approach to contract based QoS aware management of composite services. For this, the reader is referred to the overview, in the introduction, of our approach to QoS management using contracts.

4.1 Practical use of the QoS framework

Our framework of Probabilistic OrchNets developed in Section 3.4 supports soft probabilistic QoS contracts expressed as probability distributions over (possibly multi-dimensional) QoS metrics. Probability distributions can be specified either as a parameterized family of distributions, or as a finite set of

quantiles. Such contracts are part of SLA (Service Level Agreement) declaration. They can either be agreed as part of negotiation or estimated by remotely observing how a service responds in terms of QoS performance. See [39, 40] for details. The theory developed in Section 3.3–3.5 provides the needed foundations for handling monotonicity properly. Criteria ensuring monotonicity are provided. Techniques to overcome the lack of monotonicity were developed, thus providing support for managing arbitrary orchestrations. QoS aware design of composite services requires relating the QoS sub-contracts between the orchestration and its called services, and the end-to-end QoS contract between the orchestration and its clients. This task is not within the scope of this paper and the reader is referred to our previous work [39, 40], where statistical on-line detection of the violation of a probabilistic contract is also developed. On-line dynamic service selection based on QoS is a central task in QoS aware management of composite services. Procedure 1 specifying the semantics of an OrchNet offers dynamic service selection as a built-in feature.

How should we adapt an orchestration language so that it naturally supports the above concepts and techniques? This orchestration language should be enhanced with features allowing: 1) To take the proper decision based on QoS regarding competing events, actions, or service calls while executing an orchestration; key here is to identify which events, actions, or service calls are in competition when making this decision. 2) To compute the end-to-end QoS of a given execution of an orchestration by composing the QoS of the different services.

To perform the above, we only need to support the following four tasks:

Causality Tracking: Since the QoS algebra relies on the knowledge of causality relations between events, actions, or service calls, we need to *keep track of causal dependencies while executing the orchestration.*

Competition Tracking: We must *identify which events or service calls are in competition at each stage of a given execution of the orchestration.*

QoS Tracking: We need to *implement the QoS algebra with its relations and operators*
- $\oplus$ (incrementing QoS),
- $\leq$ (comparing QoS), and
- $\lhd$ (resolving competition based on QoS).

End-to-End QoS: Then, we need to be able to *compute the end-to-end QoS* of an execution of the orchestration, following Section 2.3 and Section 3.2.

Once these four tasks are supported, QoS aware management follows as a byproduct. This provides the foundations for a separation of concerns and opens the way to an orthogonal development of QoS and functional aspects of an orchestration. This important contribution is detailed next.

4.2 Weaving QoS in Orchestrations

Separation of concerns has been advocated as a recommended design discipline in the development of complex software systems. The consideration of QoS

in composite services is a source of significant increase in complexity. Tight interaction between QoS and the function performed makes QoS a crosscutting concern. Aspect Oriented Programming (AOP) has been advocated as a solution to support separation of crosscutting concerns in software development [30, 31]. In AOP, the different *aspects* are developed separately by the programmer. Their *weaving* is performed using *joinpoints* and *pointcuts*, and by having *advice* refining original pointcuts. In this section we develop a compile-time weaving of QoS aspects in composite services. Observe first that our formal model of OrchNets offers by itself support for separation of concerns in QoS management. Once the involved QoS domains have been specified with their algebraic operations, the execution policy of OrchNets (Procedure 1) entirely determines how QoS interferes with the execution of the orchestration, see the discussion of the example of Fig. 4 in Section 2.1.

Van der Aalst's WF-nets (WFnets) [2, 4, 5] are a Petri net formalism and are thus closely related to the functional part of our OrchNets. The compile-time weaving of QoS into WF-nets is best illustrated by the example of Fig. 6, where the XML-like specification explains how a functional description of a composite service can be complemented with its QoS specification. The original functional specification is BPEL-compliant and is written in **boldface**. Add-ons for QoS are written in *italics* and consist of the WSLA [29] specification of the Interface, playing the role of a rich SLA specification. Two QoS domains are declared: *RTime* (for ResponseTime) and *Cost*. These domains come up with the declaration of their associated operators following Section 2.2, namely *Cost.leq, Cost.oplus, Cost.vee, Cost.compet* and similarly for *RTime*—this is not shown on the figure since such QoS domains should be predefined and available from a library. The Interface also contains, for each called service, the declaration of the QoS measures that are relevant to it—*service1* knows only *RTime* whereas *service2* knows the two. The functional part of this specification (shown in **boldface**) collects four service calls or returns, each of which constitutes a pointcut.

The QoS-enhanced orchestration is automatically generated from the specification shown in Fig. 6—to save space, we do not show it but we only discuss the steps performed in generating it. The added code is written in roman. The first step is to initialize the metrics relevant to the orchestration, see Fig. 7. The sequence begins with the initialization of the response time carried by the token using the <assign> declaration. Concurrent invocation of the service(-) and clock = service.clock.store follow, using the <flow></flow> declaration. Once the service(-) returns, the difference between the current clock and service.clock.store is assigned to service.RTime. Resulting weaving is obtained by applying the generic rewriting rule shown on Fig. 8. The same mechanism is used for the response time of service2 and the end-to-end response time of the orchestration follows by adding the above two. Each pointcut shown in **boldface** in this figure is refined by the corresponding advice (in roman) following it.

The end-to-end evaluation of Cost for the orchestration is computed in a different way, because this kind of QoS is individually carried by the tokens rep-

```
<SLA>
  <SLAParameter name = "ResponseTime"
  type = "float" unit = "milliseconds">
        <Metric>ResponseTime</Metric>
     <Function>
        <Metric>ResponseTimeOplus</Metric>
        <Metric>ReponseTimeCompare</Metric>
        <Metric>ReponseTimeCompete</Metric>
     </Function>
  </SLAParameter>
  <SLAParameter name = "Cost"
  type = "integer" unit = "euro">
        <Metric>Cost</Metric>
     <Function>
        <Metric>CostOplus</Metric>
        <Metric>CostCompare</Metric>
        <Metric>CostCompete</Metric>
     </Function>
  </SLAParameter>
  <ServiceDefinition name="orch">
  <MetricURI http://orch.com/getMetric
   ?ResponseTime />
  <MetricURI http://orch.com/getMetric?Cost />
  </ServiceDefinition>
  <ServiceDefinition name="service1">
  <MetricURI http://service1.com/getMetric
   ?ResponseTime />
  </ServiceDefinition>
  <ServiceDefinition name="service2">
  <MetricURI http://service2.com/getMetric
   ?ResponseTime />
  <MetricURI http://service2.com/getMetric?Cost />
  </ServiceDefinition>
</SLA>

<process>
   <sequence>
      <invoke name = "service1(-)" ... />
      <receive name = "service1(-)" ... />
      <invoke name = "service2(-)" ... />
      <receive name = "service2(-)" ... />
   </sequence>
</process>
```

Fig. 6 Separation of concerns in QoS-aware specification. The functional specification is depicted last in **boldface**, whereas the QoS part is shown in *italics* on top in the form of a rich SLA specification.

resenting the queries while being processed by the orchestration. Since `Cost` is relevant to `service2` by interface declaration in Fig. 6, the call to `service2` is augmented with the return of the cost of calling `service2`. This weaving is obtained by applying the generic rewriting rule of Fig. 9. Here, the **`invoke`**

```
<assign>
   <$orch.RTime = 0 />
   <$orch.Cost = 0 />
</assign>
```

Fig. 7 Initialization step

```
<sequence>
<invoke name = "service(-)" />
...
<receive name = "service(-)" />
</sequence>
```

rewrites as:

```
<sequence>

   <flow>
      <invoke name = "service(-)" />
      <sequence>
         <invoke "clock()"/>
         <receive "clock()"
          outputVariable = "clock"/>
         <assign>
            <$service.clock.store = $clock />
         </assign>
      </sequence>
   </flow>
   ...
   <flow>
      <receive name = "service(-)" />
      <sequence>
         <invoke "clock()"/>
         <receive "clock()"
          outputVariable = "clock"/>
         <assign>
            <$service.RTime =
             $clock - $service.clock.store />
            <$orch.RTime =
             $orch.RTime + $service.RTime />
         </assign>
      </sequence>
   </flow>
  </sequence>
```

Fig. 8 Rewriting rule for weaving response time.

pointcut is not refined, only the **`receive`** is refined, by the advice code (in roman) following it.

The automatic generation of the augmented program from the original specification is a direct coding of the Procedure 1. Rules for other constructions such as the firing of a transition with several input places and the competition when a token exits a place with possible choices, are derived similarly, following Procedure 1. For general WFnets, we must keep track of the different tokens

```
<sequence>
<invoke name = "service(-)"/>
<receive name = "service(-)"/>
</sequence>
```

rewrites as:

```
<sequence>
   <invoke name = "service(-)" />
   <receive name = "service(-)"
    outputVariable = "service.Cost"  />
   <assign>
      <$orch.Cost =
       $orch.Cost + $service.Cost />
   </assign>
</sequence>
```

Fig. 9 Rewriting rule for weaving cost.

and attach QoS values to them. This amounts to keeping track of causalities between service calls that result from the WFnet. To support the weaving, pointcuts need not be explicitly declared by the programmer. They are instead obtained by pattern matching searching for keywords **`invoke`** and **`receive`** in the functional specification.

Instead of developing a tool implementing the above technique for WFnets, we have performed a prototype implementation on top of the Orc orchestration language. This is explained in the next section and subsequently illustrated using the TravelAgent2 example of Fig. 2.

4.3 Enhancing Orc for QoS

Background on Orc: Orc [22] is a general purpose language aimed to encode concurrent and distributed computations, particularly workflows and Web service orchestrations. An orchestration described in Orc is essentially an Orc expression. An Orc expression is either a *site* or is built recursively using any of the four Orc combinators. A site models any generic service which the Orc expression orchestrates. A site can be *called* with a list of parameters, and all these parameters' values have to be defined before the call can occur. A call to a site returns (or *publishes*) at most one value; it may also *halt* without returning a value. The identity site, which publishes the value x it receives as a parameter, is denoted by x (the name of its parameter). Orc allows composing service calls or actions by using a predefined small set of combinators that we describe next. In the *parallel composition* $f \mid g$, expressions f and g run in parallel. There is no direct interaction between parts of f and g and the returned values are merged by interleaving them. The *sequential composition* $f >x> g$ starts by running f. For every value v published by f, a *new* instance of g is run in parallel, with the value of x bound to v in that instance. As a particular case, $f \gg g$ performs f and then g, in sequence. The *pruning com-*

position f $<x<$ g runs f and g in parallel. When g publishes its *first* value v, the computation of g is terminated, and occurrences of x in f are replaced by v. Since f is run in parallel with g, site calls in f that have x as a parameter are blocked until g publishes a value. Finally, the *otherwise* combinator $f; g$ runs f first. If f publishes a value, g is entirely ignored. However if the computation of f halts without returning any values, then g is run. Orc also has built in sites to track passage of time (Rtime, Rwait), deal with data structures (tuples, lists, records), handle concurrency (semaphores, channels) and define new sites (class). An interested reader is referred to the Orc documentation [6] for details.

Enhancing Orc: We now describe how we integrate our QoS framework into the Orc language. In particular, we explain how we perform the four tasks listed at the end of Section 4.1 within Orc.

The ***Causality Tracking*** task consists in tracking the causal relations between execution events in the Orc interpreter. This was straightforward for WFnets, since causality is revealed by the graph structure of the net. It is not immediate for Orc programs, however. The event structure semantics of Orc [43] served as a formal specification for this. It turns out that causality can be cast into our generic algebraic framework for QoS developed in Section 2.3. Causalities are represented as pairs $x = (e, C)$, where e is the considered event and $C = \{x_1, \ldots, x_k\}$ is the set of its direct causes, recursively encoded as pairs of the same kind. The QoS domain encoding causalities is defined similarly to the QoS domain "Cost" of Section 2.2. Consequently, the generic technique developed to weave QoS into an Orc program can be instantiated to generate causalities. Details will be reported elsewhere. As a small illustration example, consider the computation of causalities for the following Orc program:

$$((2 \gg x) \; <x< \; (1 \gg 3)) \gg \mathit{print}(4)$$

We apply our generic weaving method by seeing causality as a QoS domain. We make use of two data structures in Orc : tuples, such as (f, g) and finite lists, such as $[f, g]$. The causal history is stored as a list of lists with the tuple (publication, causal past) published in the transformed program. The weaving yields the following causality-enhanced Orc program:

$$\begin{array}{l}
(\\
\quad (\\
\quad\quad ((2, []) >t> (x >(x0, _)> (x0, \mathit{union}([x], [t])))) \; <x< \\
\quad\quad ((1, []) >t> (3, [t])) \\
\quad) >t> (("\mathit{print}", [t]) >x0> (\mathit{print}(4), [x0])) \\
)
\end{array}$$

The first event has an empty causal past (represented by []). Through pattern matching, this is propagated to the next event with causal history accumulated.

[6] http://orc.csres.utexas.edu/documentation.shtml

The output of its execution yields the partial order of causes of the publication of $print(4)$:

$$4(signal, [(print, [(3, [(3, [(1, [])]), (2, [])])])])$$

Focus now on the ***Competition Tracking*** task. In its basic form, Orc does offer a way to select one publication among several candidate ones, namely by using the pruning operator. Indeed, in the Orc expression

$$f \ <x< \ (E_1 \mid E_2 \mid \cdots \mid E_n) \tag{26}$$

the first publication by E_1, E_2, ..., or E_n, preempts any future publication of the parallel composition $g \underline{\Delta} E_1 \mid E_2 \mid \cdots \mid E_n$. Since only one publication of g is picked, all possible publications of g are in mutual conflict when in the context of (26). One can regard (26) as implementing the ***Competition Tracking*** task for the particular case when the conflict is resolved on the basis of the time of occurrence of the conflicting publications, seen as a QoS measure—only the earliest one survives. We propose to lift the Orc pruning operator by resolving the conflict on the basis of an arbitrary QoS measure q given as a parameter of the generalized pruning:

$$f \ <x<_q \ (E_1 \mid E_2 \mid \cdots \mid E_n) \tag{27}$$

which is, for its definition, macro-expanded in core Orc as follows:

$$f \ <x< \ \mathbf{sort}_q(E_1 \mid E_2 \mid \cdots \mid E_n) \tag{28}$$

In (28), expression $\mathbf{sort}_q(E_1 \mid E_2 \mid \cdots \mid E_n)$ stores as a stream all publications of $(E_1 \mid E_2 \mid \cdots \mid E_n)$ upon termination and then reorders this stream according to the partial order defined by QoS measure q.[7] For the special case where QoS measure q is just the response time d, then (27) boils down to (26), the original pruning operator.

At this point, we must explain how Procedure 1 is implemented. Focus first on step 1 of that procedure, where a cluster is selected. In free choice nets, clusters localize conflicts. In core Orc, conflicts are localized in the pruning operator (26). In our extension of Orc, conflicts are localized in the QoS-based pruning operator (27). Thus, step 1 of Procedure 1 consists in selecting one among all enabled expressions of the form (27). Next, expression (27) itself is better explained with reference to Fig. 1. Fig. 1-(a) is a direct illustration of (27), whereas its equivalent form Fig. 1-(b) yields step 3 of Procedure 1 (the competition step). This discussion shows that the new feature (27) enhances Orc with a feature that is as powerful as the QoS-based conflict of our OrchNets and it explains how Procedure 1 is implemented. Details of implementing this approach are presented in Appendix B.1. We could have considered a more general feature $f \ <x<_q \ g$, where g is an arbitrary Orc expression with its several induced threads, not necessarily a parallel composition. Our current implementation does not provide this more general feature, however.

[7] Since QoS values may be partially ordered, this choice could be non-deterministic.

The ***QoS Tracking*** task of implementing the QoS algebra is handled as in the SLA declarations of Section 4.2. This is extended with *QoS Weaving* to enhance the functional declared code with a QoS enhanced output. In Appendix B we provide examples and develop the TravelAgent2/3 Examples in Orc by using the above methodology.

Finally, the ***End-to-End QoS*** task is much less obvious than for WFnets. The reason is that Orc does not handle explicitly states, transitions, and causality. Rewriting rules are needed that automatically transform functional Orc code by enhancing it for QoS, structurally. This resembles what we briefly presented regarding causality. Details will be presented elsewhere.

5 Evaluation of Our Approach

In this section, we make use of our implementation for performing contract composition, that is, estimating the end-to-end QoS of the TravelAgent2 and TravelAgent3 examples. The former is monotonic whereas the latter is not. Our study illustrates the effect of monotonicity and substantiates the need for the rich theory developed in this paper.

The experiments

Each orchestration is specified as a QoS weaved specification such as explained in Appendix B.2. For each trial, QoS values for each called service are drawn according to their specified contracts and then our automatic QoS evaluation procedure applies—we will actually use both the normal QoS evaluation from (18) and the "pessimistic" QoS evaluation from (19) and compare them. Drawing 20,000 successive trials yields, using Monte-Carlo estimation, an estimate of the end-to-end contract in the form of a probability distribution. QoS dimensions considered here are *latency, cost*, and *category*. When choices are performed according to two dimensions or more (e.g., cost and category), we make use of a weighing technique following AHP [47].

Fig. 10 displays the results of two experiments, corresponding to two different sets of contracts exposed by the called services, shown on diagrams (a) for latency, and (c) for cost. In order to evaluate the end-to-end QoS of the TravelAgent 2/3 orchestrations in a realistic setting, the AirlineCompany and HotelBooking services are modeled as distributed applications hosted on a GlassFish 3.1 server on the Inria local area network—each call to AirlineCompany or HotelBooking results in a parallel call to one of the above mentioned GlassFish applications and the corresponding latency is recorded and used for end-to-end QoS evaluation. Other services are assumed to react much quicker and are drawn from a *Student-t* distribution, not shown in the figures. Costs, on the other hand, are drawn from some Gaussian distributions (with small variance/mean ratio); note that we could as well have costs deterministic, this would not change our method.

Fig. 10 displays the estimated end-to-end QoS in diagrams (b) for latency and (d) for cost. The results are shown for both the normal QoS evaluation from (18) and the "pessimistic" QoS evaluation from (19). Not surprisingly, pessimistic evaluation yields larger end-to-end QoS estimates.

Now, recall that TravelAgent2 is monotonic, whereas TravelAgent3 is not. What are the consequences of this? In Fig. 10-right, the cost for AirlineCompany2 has been reduced as compared to Fig. 10-left. For the monotonic orchestration TravelAgent2, this reduction results in a reduction of the overall cost. For the non-monotonic orchestration TravelAgent3, however, this reduction gives raise to an *increase* in overall cost. On the other hand, pessimistic QoS evaluations are always monotonic, see Theorem 1; the results shown conform to this theorem.

Once these end-to-end measurements are taken, the negotiation of contracts and their monitoring may be done as in [39, 40]. This follows the Monte-Carlo procedure explained in [39, 40] and is thus omitted.

Discussion

When dealing with monotonic orchestrations, our contract composition procedure performs at once, both QoS evaluation and optimization. Competing alternatives are captured by the different choices occurring in the orchestration. According to Procedure 1, choice among competing alternatives is by local optimization, which implements global optimization since the orchestration is monotonic. Despite the use of Monte-Carlo simulations, this simple policy is cheaper than global optimization, even if analytic techniques are used for composing probabilistic QoS. Furthermore, when applied at run time, Procedure 1 implements late binding of services with optimal selection in a very cheap way.

Of course, there is no free lunch. If the considered orchestration is not monotonic, the above approach does not work as such, as already pointed out in [8, 11, 51], see Section 6. The bypasses developed in Section 3.5 must be used. The aggregation procedure results in aggregating services that are called in sequence, which increases granularity of the orchestration. When applied in the context of late binding, the decision is delayed until alternatives have all been explored—thus, it is hard to claim that late binding has been achieved by doing so. If pessimistic evaluation is followed, then immediate choices can be applied but, as we said, the end-to-end QoS evaluation that results is pessimistic in that the evaluation accumulates worst QoS among alternatives. So, none of the above techniques is fully satisfactory for non-monotonic orchestrations. In turn, global optimization always applies and implements best service selection—however, we question the meaning of QoS aware management when orchestrations are non-monotonic.

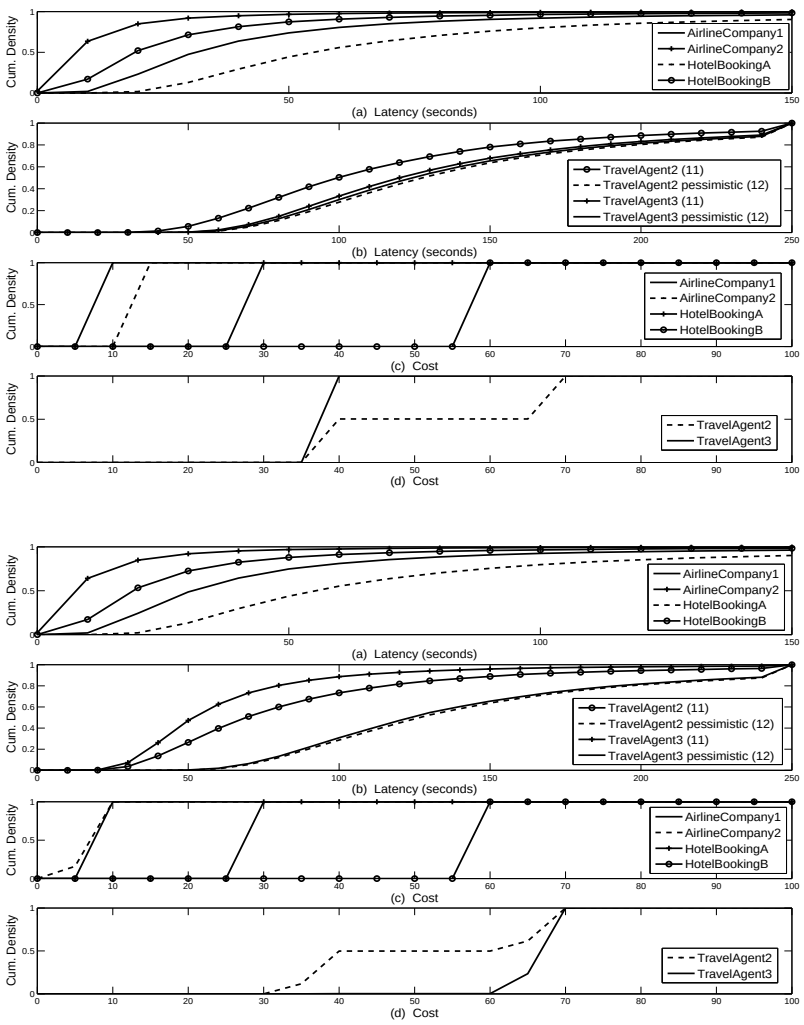

Fig. 10 We show results from two experiments (top and bottom). For each experiment we display cumulative densities of: (a) Measured latency of invoked services (b) End-to-end latency for TravelAgent 2/3 orchestrations through two evaluation schemes (c) Measured cost of invoked services (d) Returned cost invoice of TravelAgent 2/3 orchestrations.

6 Related Work

We restrict ourselves to papers dealing with QoS-aware management of composite services and addressing QoS-based design, on-line service selection, monitoring and adaptation/reconfiguration. We focus on specific papers dealing with issues relevant to our work:

- *QoS Algebraic Formulation:* While QoS composition has been studied in a variety of techniques, we are interested in mathematically sound models for QoS. We pay attention to the handling of *probabilistic* and *multi-dimensional* QoS.
- *Monotonicity:* In case of data dependent workflows, the analysis of monotonicity in design becomes crucial. We restrict our discussion to papers that have either considered this implicitly or make use of other techniques to ensure this.
- *Contracts:* Once QoS models have been specified, contractual agreements between clients and orchestrations (or similarly, between orchestrations and sub-contracted services) will need to be specified. We review some approaches that utilize the probabilistic nature of QoS to ensure mathematically sound contractual agreements.

We review the literature collected in Table 1 and Table 2, where issues of monotonicity are relevant.

We begin with the work of Yu and Bouguettaya [48]. Built-in monotonicity is still ensured, due to proper restrictions on the control flow of the considered orchestrations. We nevertheless discuss it because specific issues of interest are studied. A Service Query Algebra is proposed in which composite services are seen as graphs. They can be further composed. QoS composition is one aspect of this service composition. QoS is treated in a fully algebraic style, very much like our present approach. Probabilistic aspects are not extensively developed, however. Buscemi and Montanari (2011) [17], Buscemi and Montanari (2007) [16], De Nicola et al. (2005) [23] is a series of paper developing algebraic modeling of QoS in a way very similar to ours. By building on the seminal work of Baccelli et al. [25] on max/+-algebra, these authors develop a commutative semi-ring algebra to model QoS domains; this is almost identical to our modeling, except for our consideration of the "competition" operator used in late service binding. Then, the authors develop the cc-pi calculus to capture dynamic service binding way beyond our present study. Probabilistic frameworks are not considered, however.

The work by Bistarelli and Santini [13, 14] is discussed here because it explicitly refers to monotonicity in its title. This is, however, misleading in that this term is used in the totally different setting of "belief revision", a kind of logic in which facts can get falsified (thus the world is not monotonic in this sense).

For the next group of papers, the authors seem unaware of the issue of monotonicity for the type of orchestration they consider (we do not repeat this fact for the different papers). Cardoso et al. [20, 21] propose a predictive QoS model that allows to compute the QoS of workflows from the QoS of their atomic parts. Individual QoS measures are estimated for their minimum, maximum, and averaged values based on measurements. Rules to compute QoS composition incrementally are used (the SWR rules published in the first author's PhD), with a special attention paid to fault-tolerant systems. Probabilistic QoS is possibly supported, with, however, little technical details. The

Paper	QoS framework	Algorithms
Yu and Bouguettaya (2008) [48]	QoS parameters can be defined as "the probability of something", composition rules are proposed	Extensive study of QoS algebra; optimization of service selection by Dynamic Programming applied to the orchestration modeled as a directed graph
Bistarelli and Santini (2009, 2009, 2010) [13], [14], [12]	Probabilistic QoS supported; analytic techniques for composing component QoS to get overall service QoS	Formal language based on semirings used to aggregate QoS; however, composition rules for QoS are not detailed
Buscemi and Montanari (2011) [17], Buscemi and Montanari (2007) [16], De Nicola et al. (2005) [23]	Generic QoS is supported through a commutative semi-ring algebra; The *cc-pi calculus* is developed to model dynamic service binding with QoS-based selection and its expressiveness is studied; SLA is declared as a system of named constraints	
Cardoso et al. (2002, 2004) [20,21]	Probabilistic QoS is supported but with little details; the composition of QoS values is explained but the composition of QoS distributions is not explained	Generic formulae presented with rules for composing workflows' QoS and tested on a genome based workflow.
Hwang et al. (2004,2007) [26,27]	Probabilistic QoS is supported, with analytic techniques for QoS composition	Efficient approximations for the analytic evaluation of Probabilistic QoS composition are proposed
Menascé et al. (2008) [34]	Probabilistic QoS is supported, with analytic techniques for QoS composition, mathematical details are provided	Optimal service selection is precisely formulated and solved with an efficient heuristic

Table 1 Literature survey: Papers dealing with orchestrations allowing for a *data-dependent* workflow (thus exhibiting a risk of non-monotonicity). The issue of monotonicity is ignored, except in the work of the authors of this paper and in Ardagna et al. [11], Alrifai & Risse [8] and Zeng et al. [51], cited in Table 2, where it is identified through the discussion on global versus local optimization.

work by Hwang et al. [26, 27] is very interesting in its study of probabilistic QoS composition via analytic techniques. To avoid the computational cost resulting from state explosion in composite services, heuristic approximations are proposed. The work by Menascé et al. [34] gives a mathematically precise development of optimal service selection with cost and latency as QoS dimensions. The BPEL constructs are supported, including the "switch", which is a source of possible lack of monotonicity; alternative branches of the switch are assigned a probability. A very interesting heuristic is provided to perform near-to-optimal selection at a reasonable computational cost. The long and rich paper by Calinescu et al. [18] presents a methodology and extensive toolkit for performing QoS-based design and reconfiguration. Markov types of models are used in this toolkit, ranging from discrete and continuous Markov chains to Markov Decision Processes to deal with non-deterministic choices or data-dependent branching. QoS analyses are supported thanks to a formu-

Paper	QoS framework	Algorithms
Calinescu et al. (2011) [18]	Probabilistic QoS is supported, with analytic techniques for QoS composition (Markov models, DMC, CMC, MDP)	QoS is formally specified by using probabilistic temporal logic; extensive toolkit and model checkers are used to implement QoS-based design and reconfiguration but little detail is given about algorithms
Zeng et al. (2004, 2008, 2003) [51], [50, 52]	Probabilistic QoS is supported (restricted to Gaussian distributions); analytic techniques for QoS composition are provided	Using an integer programming formulation, global and local optimization are studied in dynamic environments and the issue of monotonicity is implicitly pinpointed
Ardagna et al.(2005) [11]; Alrifai & Risse(2009) [8]	Probabilistic QoS is not supported	QoS-aware service selection is solved via Mixed Integer Linear Programming / Multi-dimension Multi-choice 0-1 Knapsack Problem (MMKP); the issue of monotonicity is pinpointed through the comparison of local vs. global QoS guarantees
Rosario et al. (2007, 2008, 2009) [15, 39, 40]	Probabilistic QoS is supported through Soft Probabilistic contracts; Monte-Carlo simulation is proposed for QoS composition; the whole study is restricted to latency	An in-depth study of monotonicity is performed; contract composition, optimal service binding, and statistical QoS contract monitoring are developed
Rosario et al. (2009) [41, 42]	Probabilistic multi-dimensional QoS is supported, with soft Probabilistic contracts involving Monte-Carlo simulation for QoS composition	Probabilistic monotonicity is studied; a preliminary version of this paper

Table 2 Literature survey, continued.

lation using probabilistic temporal logic and associated model checkers. The methodology and toolkit reuses existing tools and did not need the development of any new engine. The paper lacks mathematical details, however, regarding the models and algorithms used.

The issue of monotonicity is identified in only three papers from our list, albeit under a different wording than ours. Ardagna et al. [11] discuss local versus global QoS guarantees and explain why optimizing QoS guarantees of local execution paths may not lead to the satisfaction of global QoS guarantees. Alrifai & Risse [8] propose a similar approach using MMKP for computationally efficient selection over global and local constraints. In Zeng et al. [51], a thorough comparison is made between local versus global optimization in service selection. It is argued that performing local optimization may not lead to optimal selection; indeed, the beginning of Section 3.2 in this paper explains exactly our example of Fig. 3. The paper explains that global optimization always provides a relevant selection, which is certainly correct. We have, how-

ever, explained in our introduction why we believe that not having monotonicity leads to a strange understanding of QoS management. Now, referring to our taxonomy, in monotonic orchestrations, local optimization is enough to ensure global optimality. Other major features of this paper are summarized in the table.

To conclude on this bibliographical study, we notice that the issue of monotonicity is mostly ignored in the literature on composite Web services, whereas it is known in the area of performance studies for general computer architectures. Our work focuses on monotonicity, its conditions, and its consequences for QoS-aware management of composite Web services.

7 Conclusion

We have studied the QoS aware management of composite services, with emphasis on QoS-based design and QoS-based on-line service selection. We have advocated the importance of monotonicity—a composite service is monotonic if a called service improving its QoS cannot decrease the end-to-end QoS of the composite service. Monotonicity goes hand-in-hand with QoS, as we think. For monotonic orchestrations, "local" and "global" optimization turn out to be equivalent. This allowed us to propose simple answers to the above tasks. Corresponding techniques are valid for both deterministic and probabilistic frameworks for QoS. We have proposed techniques to deal with the lack of monotonicity. We have observed that the issue of monotonicity has been underestimated in the literature.

To establish our approach on firm bases, we have proposed an abstract QoS calculus, whose algebra encompasses most known QoS domains so far. How QoS based design and on-line service selection are performed in our approach is formalized by the model of OrchNets. Our framework of QoS calculus and OrchNets supports multi-dimensional QoS measures, handled as partial (not total) orders. To account for high uncertainties and variability in the performance of Web services, we support probabilistic QoS.

QoS and function interfere; still, the designer expects support for separation of concerns. We provide such a support by allowing for separate SLA declaration and functional specification, followed by weaving to generate QoS-enhanced orchestrations. Our weaving techniques significantly clarifies the specification. Finally, we have proposed a mild extension of the Orc orchestration language to support the above approach—the principles of our extension could apply to BPEL [1] as well.

We believe that our approach opens new possibilities in handling orchestrations with rich QoS characteristics.

Acknowledgements The authors would like to thank Jayadev Misra and William R. Cook for fruitful discussions regarding Orc. Further thanks to the two anonymous referees for providing us with constructive comments and suggestions that have been incorporated in the revised version. This work was partially funded by the INRIA Associated Team grant

FOSSA, the ANR national research program DocFlow (ANR-06-MDCA-005) and the project CREATE ActivDoc.

A Appendix: Proofs

A.1 Proof of Theorem 2

Throughout the proof, we fix an arbitrary value ω for the daemon. We first prove the sufficiency of condition (21). Let $\mathcal{N}'$ be such that $\mathcal{N}' \geq \mathcal{N}$. Since operators $\oplus$ and $\lhd$ are both monotonic, see Definition 1, we have, by Procedure 2 and formulas (17) and (18):

$$E_\omega(\kappa(\mathcal{N}',\omega),\mathcal{N}') \geq E_\omega(\kappa(\mathcal{N}',\omega),\mathcal{N})$$

By (21) applied with $\kappa = \kappa(\mathcal{N}',\omega)$, we get that

$$E_\omega(\kappa(\mathcal{N}',\omega),\mathcal{N}) \geq E_\omega(\kappa(\mathcal{N},\omega),\mathcal{N})$$

holds. This proves the sufficiency of condition (21).

We prove necessity by contradiction. Let $(\mathcal{N},\omega,\kappa^\dagger)$ be a triple violating condition (21), in that

$$\begin{array}{l}\kappa^\dagger \text{ cannot get selected by Procedure 1, but}\\ E_\omega(\kappa^\dagger,\mathcal{N}) \geq E_\omega(\kappa(\mathcal{N},\omega),\mathcal{N}) \text{ does not hold.}\end{array}$$

Now consider the OrchNet net $\mathcal{N}' = (N, V, Q', Q_{\text{init}})$ where the family Q' is such that, $\forall t \in \kappa^\dagger$, $\xi'_t(\omega) = \xi_t(\omega)$ holds, and $\forall t \notin \kappa^\dagger$, using (5) together with the assumption that $(\mathbb{D},\leq)$ is an upper lattice, we can inductively select $\xi'_t(\omega)$ such that the following two inequalities hold:

$$\bigvee_{t\in\kappa^\dagger} q_t \leq \Big(\bigvee_{p'\in{}^\bullet t} q_{p'}\Big) \oplus \xi'_t(\omega) \tag{29}$$

$$\xi_t(\omega) \leq \xi'_t(\omega) \tag{30}$$

Condition (30) expresses that $\mathcal{N}' \geq \mathcal{N}$. By Procedure 1 defining QoS policy, (29) implies that configuration $\kappa^\dagger$ can win all competitions arising in step 3 of QoS policy, $\kappa(\mathcal{N}',\omega) = \kappa^\dagger$ holds, and thus

$$E_\omega(\kappa(\mathcal{N}',\omega),\mathcal{N}') = E_\omega(\kappa^\dagger,\mathcal{N}') = E_\omega(\kappa^\dagger,\mathcal{N})$$

However, $E_\omega(\kappa^\dagger,\mathcal{N}) \geq E_\omega(\kappa(\mathcal{N},\omega),\mathcal{N})$ does not hold, which violates monotonicity.

A.2 Proof of Theorem 3, Sufficiency

Let φ_N be the net morphism mapping U_N onto N and let $\mathcal{N}$ be any OrchNet built on U_N. We prove that condition (21) of Theorem 2 holds for $\mathcal{N}$ by induction on the number of transitions in the maximal configuration $\kappa(\mathcal{N},\omega)$ that actually occurs. The base case is when it has only one transition. Clearly this transition has minimal QoS increment and any other maximal configuration has a greater end-to-end QoS value.

Induction Hypothesis: Condition (21) of Theorem 2 holds for any maximal occurring configuration with $m-1$ transitions ($m > 1$). Formally, for an OrchNet $\mathcal{N}, \forall \omega \in \Omega, \forall \kappa \in \mathcal{V}(N)$,

$$E_\omega(\kappa, \mathcal{N}) \geq E_\omega(\kappa(\mathcal{N}, \omega), \mathcal{N}) \tag{31}$$

must hold if $|\{t \in \kappa(\mathcal{N}, \omega)\}| \leq m-1$.

Induction Argument: Consider the OrchNet $\mathcal{N}$, where the actually occurring configuration $\kappa(\mathcal{N}, \omega)$ has m transitions and let

$$\emptyset = \kappa_0(\omega) \prec \kappa_1(\omega)(= \kappa) \prec \ldots \prec \kappa_{M(\omega)}(\omega) = \kappa(\mathcal{N}, \omega)$$

be the increasing chain of configurations leading to $\kappa(\mathcal{N}, \omega)$ under QoS policy, see (3.2) — to shorten the notations, we write simply κ instead of $\kappa_1(\omega)$ in the sequel of the proof. We assume that $M(\omega) \leq m$. Let t be the unique transition such that $t \in \kappa_1(\omega)$ and set $\widehat{t} = \{t\} \cup t^\bullet$. Let κ' be any other maximal configuration of $\mathcal{N}$. Then two cases can occur.

- $t \in \kappa'$: In this case, comparing the end-to-end QoS of $\kappa(\mathcal{N}, \omega)$ and κ' reduces to comparing

  $$E_\omega(\kappa(\mathcal{N}, \omega) \setminus \widehat{t}, \mathcal{N}^\kappa) \text{ and } E_\omega(\kappa' \setminus \widehat{t}, \mathcal{N}^\kappa)$$

  where $\mathcal{N}^\kappa$ is the *future of* κ *in* $\mathcal{N} = (N, V, A, Q, Q_{\text{init}})$, obtained by replacing N by N^κ, restricting V, A, and Q to N^κ, and replacing Q_{init} by $E_\omega(\kappa, \mathcal{N})$, the QoS cost of executing configuration κ.
  Since $\kappa(\mathcal{N}, \omega) \setminus \widehat{t}$ is the actually occurring configuration in the future $\mathcal{N}^\kappa$ of transition t, using our induction hypothesis, then

  $$E_\omega(\kappa' \setminus \widehat{t}, \mathcal{N}^\kappa) \geq E_\omega(\kappa(\mathcal{N}, \omega) \setminus \widehat{t}, \mathcal{N}^\kappa)$$

  holds, which implies

  $$E_\omega(\kappa', \mathcal{N}) \;\geq\; E_\omega(\kappa(\mathcal{N}, \omega), \mathcal{N})$$

- $t \notin \kappa'$: Then there must exist a transition $t' \in \kappa'$ such that t and t' differ and belong to the same cluster $\mathbf{c}$. Hence, $\varphi_N(t)^\bullet = \varphi_N(t')^\bullet$ follows from the structural condition of Theorem 3. The futures $\mathcal{N}^\kappa$ and $\mathcal{N}^{\kappa'}$ thus are isomorphic: they only differ in the initial colors of their places. If Q_{init} and Q'_{init} are the initial QoS values for the futures $\mathcal{N}^\kappa$ and $\mathcal{N}^{\kappa'}$, then $Q_{init} \leq Q'_{init}$ holds (since $\xi_t \leq \xi_{t'}$, $t^\bullet$ has QoS lesser than $t'^\bullet$ by monotonicity of $\oplus$). On the other hand,

  $$E_\omega(\kappa(\mathcal{N}, \omega), \mathcal{N}) \;= E_\omega(\kappa(\mathcal{N}, \omega) \setminus \widehat{t}, \mathcal{N}^\kappa) \tag{32}$$

  and

  $$E_\omega(\kappa', \mathcal{N}) \;= E_\omega(\kappa' \setminus \widehat{t'}, \mathcal{N}^{\kappa'})$$

Now, since $\mathcal{N}^{\kappa'}$ and $\mathcal{N}^{\kappa}$ possess identical underlying nets and $\mathcal{N}^{\kappa'} \geq \mathcal{N}^{\kappa}$, then we get

$$E_\omega(\kappa' \setminus \widehat{t'}, \mathcal{N}^{\kappa'}) \geq E_\omega(\kappa' \setminus \widehat{t'}, \mathcal{N}^{\kappa}) \tag{33}$$

Finally, applying the induction hypothesis to (32) and using (33) yields $E_\omega(\kappa', \mathcal{N}) \geq E_\omega(\kappa(\mathcal{N}, \omega), \mathcal{N})$.

This proves that condition (21) of Theorem 2 holds and finishes the proof of the theorem.

A.3 Proof of Theorem 3, Necessity

We will show that when the structural condition of Theorem 3 is not satisfied by N, Orchnet $\mathcal{N}_N$ can violate condition (21) of Theorem 2, the necessary condition for monotonicity.

Let $\mathbf{c}$ be any cluster in U_N that violates the structural condition of Theorem 3. Since N is sound, all transitions in $\mathbf{c}$ are reachable from the initial place and so there are transitions $t_1, t_2 \in \mathbf{c}$ such that ${}^\bullet t_1 \cap {}^\bullet t_2 \neq \emptyset$, ${}^\bullet\varphi(t_1) \cap {}^\bullet\varphi(t_2) \neq \emptyset$ and $\varphi(t_1)^\bullet \neq \varphi(t_2)^\bullet$.

Define $[t] = \lceil t \rceil \setminus \widehat{t}$ and $\kappa = [t_1] \cup [t_2]$. κ is a configuration. Since $t_1^\bullet \neq t_2^\bullet$, without loss of generality, we assume that there is a place $p \in t_1^\bullet$ such that $p \notin t_2^\bullet$. Let t^* be a transition in $\mathcal{N}^\kappa$ such that $t^* \in p^\bullet$. Such a transition must exist since p can not be a maximal place: $\varphi(p)$ can not be a maximal place in N which has a unique maximal place. Now, consider the Orchnet $\mathcal{N}' > \mathcal{N}$ obtained as follows: using repeatedly condition (5) for operator $\oplus$ in Definition 1, $\xi'_{t_1}(\omega) = \xi_{t_1}(\omega), \xi'_{t_2}(\omega) \geq \xi_{t_1}(\omega)$, and, for all other $t \in \mathbf{c}, \xi'_t(\omega) \geq \xi'_{t_2}(\omega)$. For all remaining transitions of $\mathcal{N}'$, with the exception of t^*, the QoS increments are the same as that in $\mathcal{N}$ and thus are finite for ω. Finally, select $\xi'_{t^*}(\omega)$ such that

$$\xi_{t_1}(\omega) \oplus \xi'_{t^*}(\omega) > \mathbf{Q}^*(\omega) \tag{34}$$

where $\mathbf{Q}^*(\omega) \in \mathbb{D}$ will be chosen later—here we used the additional condition of Theorem 3 regarding $\mathbb{D}$, together with condition (5) for operator $\oplus$ in Definition 1. Transition t_1 has a minimal QoS increment among all transitions in cluster $\mathbf{c}$. It can therefore win the competition, thus giving raise to an actually occurring configuration $\kappa(\mathcal{N}', \omega)$. Select $\mathbf{Q}^*(\omega)$ equal to the maximal value of the end-to-end QoS of the set K of all maximal configurations κ that do not include t_1 (e.g., when t_2 fires instead of t_1). By (34), since t^* is in the future of t_1, we thus have $E_\omega(\kappa(\mathcal{N}', \omega), \mathcal{N}') \geq \xi_{t_1}(\omega) \oplus \xi'_{t^*}(\omega) > \mathbf{Q}^*(\omega) \geq E_\omega(\kappa, \mathcal{N}')$ for any configuration κ and, therefore, $\mathcal{N}'$ violates the condition (21) of Theorem 2.

A.4 Proof of Theorem 5

The proof is by contradiction. Assume that

$$\begin{array}{l}\text{there exists a pair } (\mathcal{N},\mathcal{N}') \text{ of OrchNets such that}\\ \mathcal{N} \geq \mathcal{N}' \text{ and } \mathbf{P}\{\,\omega \in \Omega \mid E_\omega(\mathcal{N}) < E_\omega(\mathcal{N}')\} > 0.\end{array} \tag{35}$$

To prove the theorem it is enough to prove that (35) implies:

$$\begin{array}{l}\text{there exists } \mathcal{N}_o, \mathcal{N}'_o \text{ such that } \mathcal{N}_o \geq \mathcal{N}'_o,\\ \text{but } E(\mathcal{N}_o) \geq^s E(\mathcal{N}'_o) \text{ does not hold}\end{array} \tag{36}$$

To this end, set $\mathcal{N}_o = \mathcal{N}$ and define $\mathcal{N}'_o$ as follows, where Ω_o denotes the set $\{\,\omega \in \Omega \mid E_\omega(\mathcal{N}) < E_\omega(\mathcal{N}')\}$:

$$\mathcal{N}'_o(\omega) = \text{ if } \omega \in \Omega_o \text{ then } \mathcal{N}'(\omega) \text{ else } \mathcal{N}(\omega)$$

Note that $\mathcal{N}_o \geq \mathcal{N}'_o \geq \mathcal{N}'$ by construction. On the other hand, we have $E_\omega(\mathcal{N}_o) < E_\omega(\mathcal{N}'_o)$ for $\omega \in \Omega_o$, and $E_\omega(\mathcal{N}_o) = E_\omega(\mathcal{N}'_o)$ for $\omega \notin \Omega_o$. By (35), we have $\mathbf{P}(\Omega_o) > 0$. Consequently, we get:

$$\begin{array}{c}[\,\forall \omega \in \Omega \;\Rightarrow\; E_\omega(\mathcal{N}_o) \leq E_\omega(\mathcal{N}'_o)\,]\\ \text{and } [\,\mathbf{P}\{\,\omega \in \Omega \mid E_\omega(\mathcal{N}_o) < E_\omega(\mathcal{N}'_o)\} > 0\,]\end{array}$$

which implies that $E(\mathcal{N}_o) \geq^s E(\mathcal{N}'_o)$ does not hold.

B Appendix: Implementation in Orc

B.1 Upgrading Orc with the bestQ operator

In order to enhance Orc with the feature presented in (27), we begin with the specification of the $\mathbf{sort}_q(E_1 \mid E_2 \mid \cdots \mid E_n)$ which sorts the list of QoS values from a domain q. The *merge sort* algorithm is employed here to sort through the list of QoS values according to the specified partial order po.

```
def mergeBy[A](lambda (A,A) :: Boolean, List[A], List[A]) :: List[A]
def mergeBy(po, xs, []) = xs
def mergeBy(po, [], ys) = ys
def mergeBy(po, x:xs, y:ys) = if po(y,x) then y:mergeBy(po,x:xs,ys)
  else x:mergeBy(po,xs,y:ys)

def sortQ[A](lambda (A,A) :: Boolean, List[A]) :: List[A]
def sortQ(po, []) = []
def sortQ(po, [x]) = [x]
def sortQ(po, xs) =
  val half = Floor(length(xs)/2)
  val front = take(half, xs)
  val back = drop(half, xs)
  mergeBy(po, sortBy(po, front), sortQ(po, back))
```

To complete the implementation of f <x< $\mathbf{sort}_q(E_1 \mid E_2 \mid \cdots \mid E_n)$, the head of the sorted list is selected. This results in the competition winner presented as the output of the bestQ operator ((27)).

```
def head[A](List[A]) :: A
def head(x:xs) = x

def bestQ[A](lambda (A,A) :: Boolean, List[A]) :: A
def bestQ(comparer, publisher) = head(sortBy(comparer, publisher))
```

The bestQ operator performs the function of the competition operator as in Step 3 of the QoS policy. In trivial cases, this yields $q \lhd_q \{q_1, q_2, \ldots, q_n\} = q$ (bestQ output), with q winning the competition according to the specified partial order $\leq_q$ (po).

B.2 QoS Weaving

These are the implementation steps to enhance functional Orc specifications with QoS. Refer to the informal specification of the TravelAgent2 orchestration in Section 1.

1. *SLA / QoS Declaration*: A library of pre-defined QoS classes that specify the types, domains, operations and units for various metrics. This makes use of the `class` construct in Orc that provides the capability to implement sites within Orc. This can include a variety of QoS domains (Latency, Cost, Security, Reliability, Throughput) that can be instantiated and reused when required. Note that multiple units may be specified: eg. seconds and milliseconds for latency/throughput, cost in items/currencies. As the classes are declared within Orc, the classes/definitions may be updated when required. For the TravelAgent2/3 example, domains such as *Latency* and *Cost* are relevant. An example is shown for the domains of Latency and Cost as follows.

```
def bestQoS(comparer,publisher) = head(sortBy(comparer,publisher))
def addNum(x, y) = x + y
def zipWith(_, [], _) = []
def zipWith(_, _, []) = []
def zipWith(f, x:xs, y:ys) = f(x, y) : zipWith(f, xs, ys)

-- Latency.inc -- Types and Definitions for Response Time
def LatencyIncrement(sitex) =
  {- Using tuple construction as fork-join -}
  (sitex, Rtime()) >(sitex, st)> (sitex, Rtime()-st)
type TimeUnit = Second() | Millisecond()

def class ResponseTime(unit) =
  def QoS(sitex) =
    LatencyIncrement(sitex) >(_, q)>
    (Ift(unit = Millisecond()) >> q | Ift(unit = Second()) >>
        q/1000)
  def QoSOplus(rt1, rt2) = rt1+rt2
  def QoSCompare(rt1, rt2) = rt1 <= rt2
```

```
  def QoSCompete(rt1, rt2) = bestQoS(QoSCompare, [rt1, rt2])
  def QoSVee(rt1, rt2) = max(rt1, rt2)
  stop


-- Cost.inc -- Types and Definitions for Cost
type CostUnit = Items() | CurrencyEuros() | CurrencyDollars()
def CostValue() = 1
def class Cost(unit) =
  def UnitConverter(x, Items()) = x
  def UnitConverter(x, CurrencyEuros()) = x*100
  def UnitConverter(x, CurrencyDollars()) = x*80
  def QoS(sitex, c) =
    val s = Ref([])
    signal >> (s? >q> (if q=[] then s := c >> s? else QoSOplus(s?,
        c) >v> s := v >> UnitConverter(s?, unit)))
  def QoSOplus(c1, c2) = zipWith(addNum, c1, c2)
  def QoSCompare(c1, c2) = foldl((&&), true, zipWith((<=), c1, c2))
  def QoSCompete(c1, c2) = bestQoS(QoSCompare, [c1, c2])
  def QoSVee(c1, c2) = zipWith(max, c1, c2)
  stop
```

In the above declaration, multiple data structures such as records $\{..\}$, lists $[f, g]$ and tuples (f, g) are used. Other general sites available in Orc such as real time (`Rtime`) rewritable storage locations (`Ref` and FIFO channels (`Channel`) are also invoked.

Once these classes are declared, the SLA type checker checks for Ambient/Non-ambient QoS types before passing the control flow. If the correct typing is not found, the SLA site blocks further evaluation. A non-ambient metric (eg. cost) has competition operator defined *trivially*. However, for ambient metrics (eg. latency), the competition operator must be specified explicitly when combined with non-ambient metrics. This would specify whether the lexicographic ordering implies a infemum/supremum for the ambient metrics.

```
-- SLA.inc
include "Latency.inc"
include "Cost.inc"

def class NonAmbient(QoStype) =
  def QoSCompete(Number,Number) :: Number
  def QoSCompete(q1,q2) = head(sortBy((<:), [q1,q2]))
  signal

def class Ambient(QoStype,competition) =
  def QoSCompete(Number,Number) :: Number
  def QoSCompete(q1,q2) = head(sortBy(competition,  [q1,q2]))
  signal
```

2. *QoS Registry*: This registers services with relevant QoS classes, QoS metric units and specific handles for accessing them. The registry is defined using the `records` data structure that can match keys to a record pattern. By providing multiple QoS units, it is possible to re-use the same class of QoS metrics multiple times by the same set of sites. Note that handles are also specified – additional information that must be returned by the service in

order to satisfy the QoS requirements. Instances of handles include cost increments (items/currency), latency increments (milliseconds/seconds) and security levels that need to be specified. As the orchestration requires these increments to generate the end-to-end QoS increment for each domain, the sites must imperatively provide these handles. A site can also be neutral (zero increment) to certain domains.
An example is provided with three sites `TAgent`, `Airline` and `Hotel` specified with various QoS domains, units and handles. An additional `QoSMatch` site matches the site identifier with these values when invoked.

```
-- QoSRegistry.inc
val QoSRegistry =
 [
  {. name = "TAgent", QoSDom = ResponseTime, QoSUnit = Millisecond,
       Handle = LatencyIncrement .},
  {. name = "TAgent", QoSDom = Cost, QoSUnit = CurrencyEuros,
       Handle = CostValue .},
  {. name = "Airline", QoSDom = Cost, QoSUnit = Items,
       Handle = CostValue .},
  {. name = "Hotel", QoSDom = Cost, QoSUnit = Items,
       Handle = CostValue  .}
 ]

def QoSMatch(siteID) = each(QoSRegistry) >M> Ift(M.name = siteID)
  >> (M.QoSDom,M.QoSUnit,M.Handle)
```

3. *Validating Registry Entries*: As the registry contains many services with possibly conflicting QoS domains, accurate mapping of service and domains may be needed. This can be done either *permissively* (not specified domains produce zero increments) or *strictly* (restricting QoS domains according to the mapping).
For a Orc expression **def** `f() = (g1(),...gN())` (service `f` invoking `gi`), an injective partial function is defined as $\mathrm{QoS}(f) \mapsto \mathrm{QoS}(g_i)$. Note that this definition allows for multiple instances of QoS classes to be defined for each of these services. A `QoSValidate` site is implemented that checks for strict conformance of QoS domains between the caller/callee sites.

```
def QoSValidate(callersiteID,caleesiteID) =
 (collect(defer(QoSMatch,callersiteID)),
 collect(defer(QoSMatch,caleesiteID)))
 >(A,B)> ( Ift(A.QoSDom = B.QoSDom) >> signal
 | Iff(A.QoSDom = B.QoSDom) >> Println("Registry Entries Missing")
 >> stop)
```

4. *QoS Weaving*: The QoS Weaver site weaves the values generated by the sites (with appropriate domains and handles) and generates the tuple of `Data, QoS`. Note that the check for the domains and handles are strict with computation stopped otherwise. For Ambient metrics (eg. `Latency`) the competition operator must be specified.

```
-- QoSWeaver.inc
def QoSWeaver(site,(lookup,unit,handle)) =

   def ResponseTimeCheck(competition) =
```

```
      Ift(lookup = ResponseTime && handle = LatencyIncrement) >>
      (Ambient(ResponseTime,competition)
      >> (ResponseTime(unit).QoS(site)))
      ; stop

     def CostCheck() =
      Ift(lookup = Cost && handle = CostValue) >>
      (NonAmbient(Cost) >> Cost(unit).QoS(site,CostValue()))
      ; stop

signal >> v<v<(ResponseTimeCheck(max)| CostCheck())

def QoS(site,identifier) =
   val Data = Ref()
   def QoSCollect(v) = collect(defer2(QoSWeaver,Data?,v))
site >d> Data:=d >> collect(defer(QoSMatch,identifier)) >v>
 (Data?, map(QoSCollect,v))
```

The weaver also incorporates a `QoS` site that wraps a specific site with the QoS increment produced by it.

5. *Functional Declaration*: Once these steps are completed, the Orc expression may be written after including the necessary sites (QoS Declarations, Registries). The site declarations must specify the site identifiers that would be invoked from the registry.

```
-- Site Definitions
def class AirlineCompany() =
  def function(GenerateInvoice) =
   bestQ(compareCost, defer(inquireCost,AirlineList))
   >q> GenerateInvoice >> GenerateInvoice.AirQuote := q
  def QoSID() = "Airline"
  stop

def class HotelBooking() =
  def function(GenerateInvoice) =
   bestQ(compareCategory, defer(inquireCategory,HotelList))
   >q> GenerateInvoice >> GenerateInvoice.HotelQuote := q
  def QoSID() = "Hotel"
  stop

def class TravelAgent() =
  def function(salesOrder,budget) = timeout(
   (acceptOrder(salesOrder, budget) >(bookingRequest, budget)>
   quoteAirfare(bookingRequest) >> quoteHotel(bookingRequest) >>
   checkBudget(bookingRequest, budget)), timeoutVal, salesOrder)
   >Some(bookingRequest)> bookingRequest
  def QoSID() = "TAgent"
  stop

def QoSsite(sitex) = QoS(sitex.function(),sitex.QoSID())
```

Once the sites have been declared, the goal expression may be written with the relevant classes included. The QoS weaving automatically equips relevant sites with their QoS increments. Note that the functional declarations can make use of the QoS outputs as well.

B.3 TravelAgent2 Example

The QoS-weaved output of the TravelAgent2 orchestration is provided with the original functional specification in (**bold**) and the QoS specification in roman. Increments to domains Cost and ResponseTime are accumulated as the control flow progresses in the orchestration. The code of QoS enhanced output is installed on the Orc site at url http://orc.csres.utexas.edu/papers/qos-aware.shtml, from where it can be run.

```
--SLA Declaration
def bestQoS(comparer,publisher) = head(sortBy(comparer,publisher))
def addNum(x, y) = x + y
def zipWith(_, [], _) = []
def zipWith(_, _, []) = []
def zipWith(f, x:xs, y:ys) = f(x, y) : zipWith(f, xs, ys)

-- Types and Definitions for Response Time
def LatencyIncrement(sitex) =
  - Using tuple construction as fork-join -
  (sitex, Rtime()) >(sitex, st)> (sitex, Rtime()-st)
type TimeUnit = Second() | Millisecond()

def class ResponseTime(unit) =
  def QoS(sitex) =
    LatencyIncrement(sitex) >(_, q)>
    (Ift(unit = Millisecond()) >> q | Ift(unit = Second()) >> q/1000)
  def QoSOplus(rt1, rt2) = rt1+rt2
  def QoSCompare(rt1, rt2) = rt1 <= rt2
  def QoSCompete(rt1, rt2) = bestQoS(QoSCompare, [rt1, rt2])
  def QoSVee(rt1, rt2) = max(rt1, rt2)
  stop

-- Types and Definitions for Cost
type CostUnit = Items() | CurrencyEuros() | CurrencyDollars()
def CostValue() = 1
def class Cost(unit) =
  def UnitConverter(x, Items()) = x
  def UnitConverter(x, CurrencyEuros()) = x*100
  def UnitConverter(x, CurrencyDollars()) = x*80
  def QoS(sitex, c) =
    val s = Ref([])
    signal >> (s? >q> (if q=[] then s := c >> s?
     else QoSOplus(s?, c) >v> s := v >> UnitConverter(s?, unit)))
  def QoSOplus(c1, c2) = zipWith(addNum, c1, c2)
  def QoSCompare(c1, c2) = foldl((&&), true, zipWith((<=), c1, c2))
  def QoSCompete(c1, c2) = bestQoS(QoSCompare, [c1, c2])
  def QoSVee(c1, c2) = zipWith(max, c1, c2)
  stop

def class NonAmbient(QoStype) =
  def QoSCompete(Number,Number) :: Number
  def QoSCompete(q1,q2) = head(sortBy((<:), [q1,q2]))
  signal

def class Ambient(QoStype,competition) =
  def QoSCompete(Number,Number) :: Number
```

```
  def QoSCompete(q1,q2) = head(sortBy(competition,  [q1,q2]))
  signal

--QoS Registry
val QoSRegistry =
 [
  . name = "TAgent", QoSDom = ResponseTime, QoSUnit = Millisecond,
Handle = LatencyIncrement .,
  . name = "TAgent", QoSDom = Cost, QoSUnit = CurrencyEuros,
Handle = CostValue .,
  . name = "Airline", QoSDom = Cost, QoSUnit = Items,
Handle = CostValue .,
  . name = "Hotel", QoSDom = Cost, QoSUnit = Items,
Handle = CostValue  .
 ]

def QoSMatch(siteID) = each(QoSRegistry) >M> Ift(M.name = siteID)
  >> (M.QoSDom,M.QoSUnit,M.Handle)

def QoSValidate(callersiteID,caleesiteID) =
 (collect(defer(QoSMatch,callersiteID)),
 collect(defer(QoSMatch,caleesiteID)))
 >(A,B)> ( Ift(A.QoSDom = B.QoSDom) >> signal
 | Iff(A.QoSDom = B.QoSDom) >> Println("Registry Entries Missing")
 >> stop)

--QoS Weaving
def QoSWeaver(site,(lookup,unit,handle)) =
  def ResponseTimeCheck(competition) =
    Ift(lookup = ResponseTime && handle = LatencyIncrement) >>
     (Ambient(ResponseTime,competition)
     >> (ResponseTime(unit).QoS(site)))
     ; stop

  def CostCheck() =
    Ift(lookup = Cost && handle = CostValue) >>
     (NonAmbient(Cost) >> Cost(unit).QoS(site,CostValue()))
     ; stop
  signal >> v<v<(ResponseTimeCheck(max)| CostCheck())

def QoS(site,identifier) =
  val Data = Ref()
  def QoSCollect(v) = collect(defer2(QoSWeaver,Data?,v))
   site >d> Data:=d >> collect(defer(QoSMatch,identifier)) >v>
   (Data?, map(QoSCollect,v))

--TravelAgent2 Example
val AirlineList = ["Airline 1", "Airline 2"]
val HotelList = ["Hotel A", "Hotel B"]
def QoSsite(sitex) = QoS(sitex.function(),sitex.QoSID())
def bestQ(comparer,publisher) =  head(sortBy(comparer,collect(publisher)))

--Simulation utilities
def cat() = if (Random(1)=1) then "Economy" else "Premium"
val simElaspedTime = Rclock()

--Functional Sites declared
def GenerateOrder(SalesOrder,Budget) = Dictionary() >GenerateInvoice>
```

```
  GenerateInvoice.TravelAgent := SalesOrder.ordernumber? >>
   GenerateInvoice.acceptedTime := simElaspedTime.time() >>
   Println("Order "+GenerateInvoice.TravelAgent?+" accepted at time "
   +GenerateInvoice.acceptedTime?) >> (GenerateInvoice,Budget)

def inquireCost(List) = each(List) >sup> Dictionary() >ProductDetails>
  ProductDetails.Company := sup>> ProductDetails.cost := Random(50)
   >> ProductDetails
def inquireCategory(List) = each(List) >sup> Dictionary() >ProductDetails>
  ProductDetails.Company := sup >> ProductDetails.cost := Random(50) >>
   ProductDetails.category := cat() >> ProductDetails

def compareCost(x, y) = x.cost? <= y.cost?
def compareCategory(x, y) = if x.category?="Economy" then false
  else if y.category?="Economy" then true else compareCost(x, y)

def CheckBudget(GenerateInvoice,Budget) =
  if (GenerateInvoice.AirQuote?.cost? + GenerateInvoice.HotelQuote?.cost?
   <: Budget) then GenerateInvoice else
   (Println("Resubmit Order " +GenerateInvoice.TravelAgent?) >>
   Dictionary() >SalesOrder>
   SalesOrder.ordernumber:= GenerateInvoice.TravelAgent?
   >> (SalesOrder,GenerateOrder(SalesOrder,Budget)))
def timeout(x, t, SalesOrder) = Let(Some(x)
  | (Rwait(t) >> notifyFail(SalesOrder, "Timeout") >> None()))
def notifyFail(SalesOrder, reaSalesOrdern) =
  Println("Order "+SalesOrder.id?+" failed: "+reaSalesOrdern)
   >> stop

--QoS-Aware Sites
def class AirlineCompany(GenerateInvoice,Cost) =
  def function() = bestQ(compareCost, defer(inquireCost,AirlineList))
   >q> GenerateInvoice >> GenerateInvoice.AirQuote := q
  def QoSID() =  "Airline"
  stop

def class HotelBooking(GenerateInvoice,Cost) =
  def function() = bestQ(compareCategory, defer(inquireCategory,HotelList))
   >q> GenerateInvoice >> GenerateInvoice.HotelQuote := q
  def QoSID() =  "Hotel"
  stop

def class TravelAgent(SalesOrder,Budget,Cost,Latency) =
  def function() =timeout((GenerateOrder(SalesOrder,Budget)
  >(GenerateInvoice,Budget)> QoSsite(AirlineCompany(GenerateInvoice,Cost))
   >(_,AirQoS)> GenerateInvoice.AirQoS := AirQoS
   >> QoSsite(HotelBooking(GenerateInvoice,Cost)) >(_,HotelQoS)>
   GenerateInvoice.HotelQoS := HotelQoS
   >> CheckBudget(GenerateInvoice,Budget)) , 10000, SalesOrder)
   >Some(GenerateInvoice)> GenerateInvoice
  def QoSID()= "TAgent"
  stop

--Simulation
def simulateOrders(50) = stop
def simulateOrders(n) = Dictionary() >SalesOrder> SalesOrder.ordernumber:= n
  >> SalesOrder
  | Rwait(Random(100)) >> simulateOrders(n+1)
```

```
simulateOrders(1) >SalesOrder>
  QoSsite(TravelAgent(SalesOrder,150,Cost,ResponseTime))
  >(GenerateInvoice,QoS)> ((GenerateInvoice.TravelAgent?,QoS)
   >> Println("Invoice for order "+SalesOrder.ordernumber?+"
   presented at time "+simElaspedTime.time())
   >> (GenerateInvoice.AirQoS?,GenerateInvoice.HotelQoS?)
   >([[aq]],[[hq]])> (aq,hq)
   >> (GenerateInvoice.TravelAgent?,QoS))
```

References

1. Web Services Business Process Execution Language Version 2.0. Tech. rep., OASIS Standard, Available from: http://docs.oasisopen.org/wsbpel/2.0/wsbpel-v2.0.pdf (April, 2007)
2. van der Aalst, W.M.P.: Verification of workflow nets. In: ICATPN, pp. 407–426 (1997)
3. van der Aalst, W.M.P.: The Application of Petri Nets to Workflow Management. The Journal of Circuits, Systems and Computers **8**(1), 21–66 (1998). URL citeseer.ist.psu.edu/vanderaalst98application.html
4. van der Aalst, W.M.P., van Hee, K.M.: Workflow Management: Models, Methods, and Systems. MIT Press (2002)
5. van der Aalst, W.M.P., ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P.: Workflow patterns. Distrib. Parallel Databases **14**(1), 5–51 (2003). DOI http://dx.doi.org/10.1023/A:1022883727209
6. Abundo, M., Cardellini, V., Presti, F.L.: Optimal admission control for a qos-aware service-oriented system. In: ServiceWave, pp. 179–190 (2011)
7. Alain Bensoussan: Stochastic Control of Partially Observable Systems. Cambridge University Press (1992)
8. Alrifai, M., Risse, T.: Combining global optimization with local selection for efficient qos-aware service composition. In: WWW, pp. 881–890 (2009)
9. Ardagna, D., Ghezzi, C., Mirandola, R.: Model Driven QoS Analyses of Composed Web Services. In: P. Mähönen, K. Pohl, T. Priol (eds.) ServiceWave, *Lecture Notes in Computer Science*, vol. 5377, pp. 299–311. Springer (2008)
10. Ardagna, D., Giunta, G., Ingraffia, N., Mirandola, R., Pernici, B.: QoS-Driven Web Services Selection in Autonomic Grid Environments. In: R. Meersman, Z. Tari (eds.) OTM Conferences (2), *Lecture Notes in Computer Science*, vol. 4276, pp. 1273–1289. Springer (2006)
11. Ardagna, D., Pernici, B.: Global and Local QoS Guarantee in Web Service Selection. In: C. Bussler, A. Haller (eds.) Business Process Management Workshops, vol. 3812, pp. 32–46 (2005)
12. Bistarelli, S., Montanari, U., Rossi, F., Santini, F.: Unicast and multicast qos routing with soft-constraint logic programming. ACM Trans. Comput. Logic **12**(1), 5:1–5:48 (2010)
13. Bistarelli, S., Santini, F.: A nonmonotonic soft concurrent constraint language for sla negotiation. Electr. Notes Theor. Comput. Sci. **236**, 147–162 (2009)
14. Bistarelli, S., Santini, F.: Soft constraints for quality aspects in service oriented architectures. In: Young Researchers Workshop on Service-Oriented Computing, pp. 51–65 (2009)
15. Bouillard, A., Rosario, S., Benveniste, A., Haar, S.: Monotonicity in Service Orchestrations. In: G. Franceschinis, K. Wolf (eds.) Petri Nets, *Lecture Notes in Computer Science*, vol. 5606, pp. 263–282. Springer (2009)
16. Buscemi, M.G., Montanari, U.: Cc-pi: a constraint-based language for specifying service level agreements. In: Proceedings of the 16th European conference on Programming, ESOP'07, pp. 18–32. Springer-Verlag (2007). URL http://dl.acm.org/citation.cfm?id=1762174.1762179

17. Buscemi, M.G., Montanari, U.: Qos negotiation in service composition. J. Log. Algebr. Program. **80**(1), 13–24 (2011)
18. Calinescu, R., Grunske, L., Kwiatkowska, M., Mirandola, R., Tamburrelli, G.: Dynamic QoS Management and Optimization in Service-Based Systems. IEEE Transactions on Software Engineering **37**(3), 387 –409 (2011)
19. Cardellini, V., Casalicchio, E., Grassi, V., Presti, F.L.: Adaptive management of composite services under percentile-based service level agreements. In: ICSOC 2010, LNCS 6470, pp. 381–395 (2010)
20. Cardoso, J., Sheth, A.P., Miller, J.A.: Workflow Quality of Service. In: K. Kosanke, R. Jochem, J.G. Nell, A.O. Bas (eds.) ICEIMT, *IFIP Conference Proceedings*, vol. 236, pp. 303–311. Kluwer (2002)
21. Cardoso, J., Sheth, A.P., Miller, J.A., Arnold, J., Kochut, K.: Quality of Service for workflows and Web service processes. J. Web Sem. **1**(3), 281–308 (2004)
22. Cook, W.R., Patwardhan, S., Misra, J.: Workflow Patterns in Orc. In: Coordination, pp. 82–96 (2006)
23. De Nicola, R., Ferrari, G., Montanari, U., Pugliese, R., Tuosto, E., Jacquet, J.M.: Coordination Models and Languages, chap. A Process Calculus for QoS-Aware Applications, pp. 246–252. Springer Berlin / Heidelberg (2005)
24. Esparza, J., Römer, S., Vogler, W.: An improvement of McMillan's Unfolding Algorithm. Formal Methods in System Design **20**(3), 285–310 (2002)
25. F. Baccelli and G. Cohen and G.J. Olsder and J-P. Quadrat: Synchronization and Linearity. Wiley Series in Probability and Mathematical Statistics, John Wiley (1992)
26. Hwang, S.Y., Wang, H., Srivastava, J., Paul, R.A.: A Probabilistic QoS Model and Computation Framework for Web Services-Based Workflows. In: ER, pp. 596–609 (2004)
27. Hwang, S.Y., Wang, H., Tang, J., Srivastava, J.: A probabilistic approach to modeling and estimating the QoS of web-services-based workflows. Inf. Sci. **177**(23), 5484–5503 (2007)
28. Kattepur, A.: Importance sampling of probabilistic contracts in web services. In: G. Kappel, Z. Maamar, H.R. Motahari-Nezhad (eds.) ICSOC, *Lecture Notes in Computer Science*, vol. 7084, pp. 557–565. Springer (2011)
29. Keller, A., Ludwig, H.: The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services. J. Network Syst. Manage. **11**(1) (2003)
30. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., marc Loingtier, J., Irwin, J.: Aspect-oriented programming. In: ECOOP. SpringerVerlag (1997)
31. Kiselev, I.: Aspect-Oriented Programming with AspectJ. Sams, Indianapolis, IN, USA (2002)
32. Kitchin, D., Cook, W.R., Misra, J.: A Language for Task Orchestration and its Semantic Properties. In: Proc. of the Intl. Conf. on Concurrency Theory (CONCUR) (2006)
33. Marsan, M.A., Balbo, G., Bobbio, A., Chiola, G., Conte, G., Cumani, A.: The effect of execution policies on the semantics and analysis of stochastic petri nets. IEEE Trans. Software Eng. **15**(7), 832–846 (1989)
34. Menascé, D.A., Casalicchio, E., Dubey, V.K.: A heuristic approach to optimal service selection in Service Oriented Architectures. In: A. Avritzer, E.J. Weyuker, C.M. Woodside (eds.) WOSP, pp. 13–24. ACM (2008)
35. Misra, J., Cook, W.R.: Computation Orchestration: A Basis for Wide-Area Computing. Journal of Software and Systems Modeling **May** (2006). Available for download at `http://dx.doi.org/10.1007/s10270-006-0012-1`
36. Moshe Shaked and J. George Shanthikumar: Stochastic Orders and their Applications. Academic Press (1994)
37. Moshe Shaked and J. George Shanthikumar: Stochastic Orders. Springer (2007)
38. Murata, T.: Petri nets: Properties, analysis and applications. In: Proceedings of the IEEE, vol. 77, pp. 541–580 (1989)
39. Rosario, S., Benveniste, A., Haar, S., Jard, C.: Probabilistic qos and soft contracts for transaction based web services. In: ICWS, pp. 126–133. IEEE Computer Society (2007)
40. Rosario, S., Benveniste, A., Haar, S., Jard, C.: Probabilistic QoS and soft contracts for transaction based Web services orchestrations. IEEE Transactions on Service Computing **1**(4) (2008)

41. Rosario, S., Benveniste, A., Jard, C.: A Theory of QoS for Web Service Orchestrations. Research Report RR-6951, INRIA (2009). Available from http://hal.inria.fr/inria-00391592/PDF/RR-6951.pdf
42. Rosario, S., Benveniste, A., Jard, C.: Flexible probabilistic qos management of transaction based web services orchestrations. In: ICWS, pp. 107–114. IEEE (2009)
43. Rosario, S., Kitchin, D., Benveniste, A., Cook, W.R., Haar, S., Jard, C.: Event structure semantics of orc. In: M. Dumas, R. Heckel (eds.) WS-FM, *Lecture Notes in Computer Science*, vol. 4937, pp. 154–168. Springer (2007)
44. Sato, N., Trivedi, K.S.: Stochastic modeling of composite web services for closed-form analysis of their performance and reliability bottlenecks. In: B.J. Krämer, K.J. Lin, P. Narasimhan (eds.) ICSOC, *Lecture Notes in Computer Science*, vol. 4749, pp. 107–118. Springer (2007)
45. Segala, R., Lynch, N.A.: Probabilistic simulations for probabilistic processes. In: B. Jonsson, J. Parrow (eds.) CONCUR, *Lecture Notes in Computer Science*, vol. 836, pp. 481–496. Springer (1994)
46. Teturo Kamae and Ulrich Krengel and George L. O'Brien: Stochastic inequalities on partially ordered spaces. The Annals of Probability **5(6)**, 899–912 (1977)
47. Thomas L. Saaty: How to make a decision: the Analytic Hierarchy Process. European Jounral of Operational Research **48**(2), 9–26 (1990)
48. Yu, Q., Bouguettaya, A.: Framework for Web service query algebra and optimization. TWEB **2**(1) (2008)
49. Yu, T., Lin, K.J.: Service Selection Algorithms for Composing Complex Services with Multiple QoS Constraints. In: B. Benatallah, F. Casati, P. Traverso (eds.) ICSOC, *Lecture Notes in Computer Science*, vol. 3826, pp. 130–143. Springer (2005)
50. Zeng, L., Benatallah, B., Dumas, M., Kalagnanam, J., Sheng, Q.Z.: Quality driven Web services composition. In: WWW, pp. 411–421 (2003)
51. Zeng, L., Benatallah, B., Ngu, A.H.H., Dumas, M., Kalagnanam, J., Chang, H.: QoS-Aware Middleware for Web Services Composition. IEEE Trans. Software Eng. **30**(5), 311–327 (2004)
52. Zeng, L., Ngu, A.H.H., Benatallah, B., Podorozhny, R.M., Lei, H.: Dynamic composition and optimization of Web services. Distributed and Parallel Databases **24**(1-3), 45–72 (2008)
53. Zheng, H., Yang, J., Zhao, W., Bouguettaya, A.: Qos analysis for web service compositions based on probabilistic qos. In: G. Kappel, Z. Maamar, H. Motahari-Nezhad (eds.) Service-Oriented Computing, *Lecture Notes in Computer Science*, vol. 7084, pp. 47–61. Springer Berlin / Heidelberg (2011)